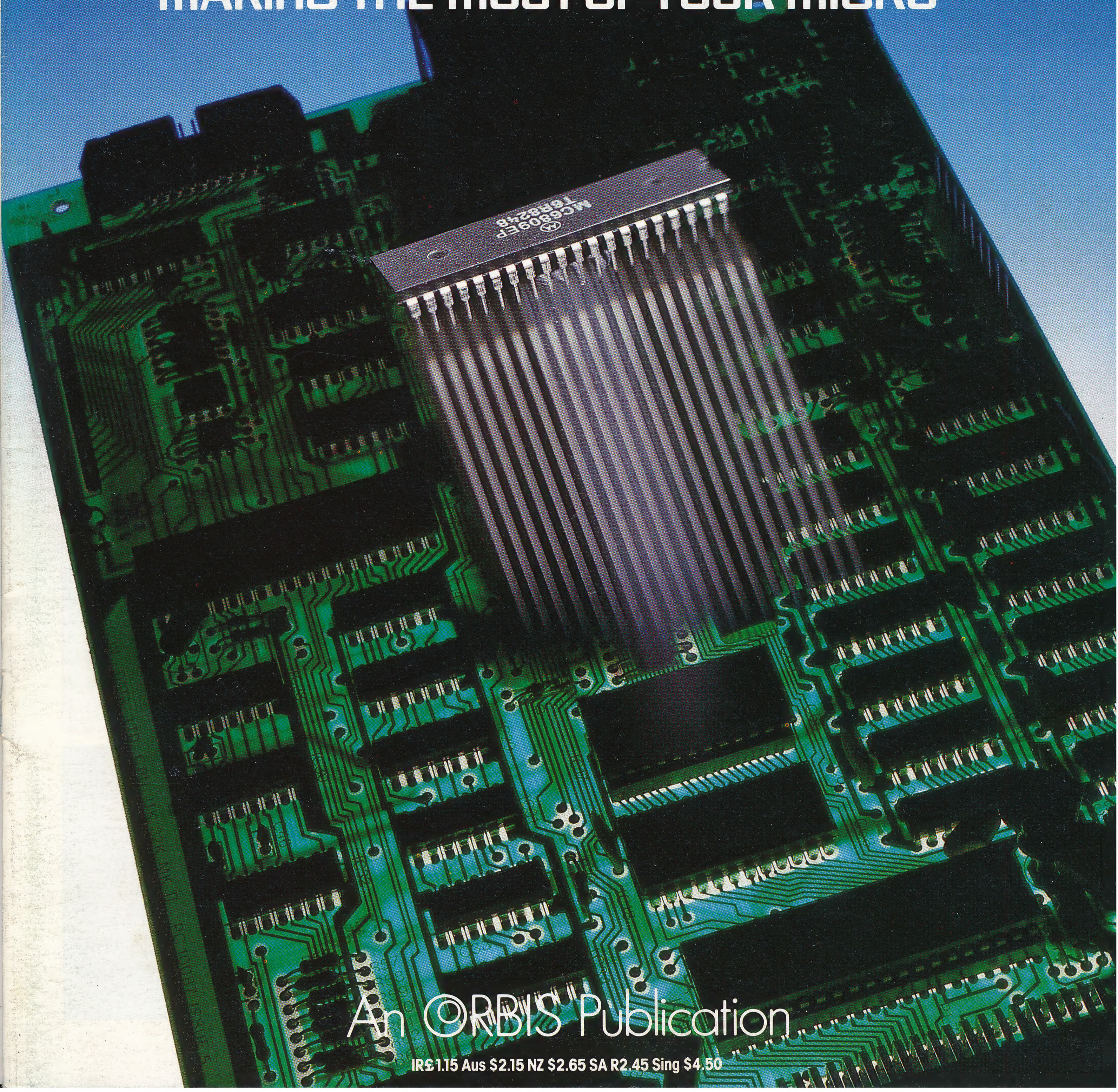
ISSN 0265-2919

90p **84**

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO



An ORBIS Publication

IR£1.15 Aus $2.15 NZ $2.65 SA R2.45 Sing $4.50

CONTENTS

APPLICATION

FORMULA FOR SUCCESS A look at how a probability theorem such as Bayes' rule can be applied to forecasting problems 1672

HARDWARE

TOURING THE CIRCUIT We begin a short series looking at the components of the printed circuit boards of various micros 1661

SOFTWARE

THE DOG AND BUCKET The interactive character adventure game listing is presented in its entirety 1674

JEWEL OF A SYSTEM A new series begins on Digital Research's GEM 1670

COMPUTER SCIENCE

COMMON USAGE COBOL is the most widely used computer language for business and commercial applications 1663

JARGON

FROM STANDARDISATION TO STREAM A weekly glossary of computing terms 1669

PROGRAMMING PROJECTS

SPREADING OUT Several new functions are added to our spreadsheet program 1666

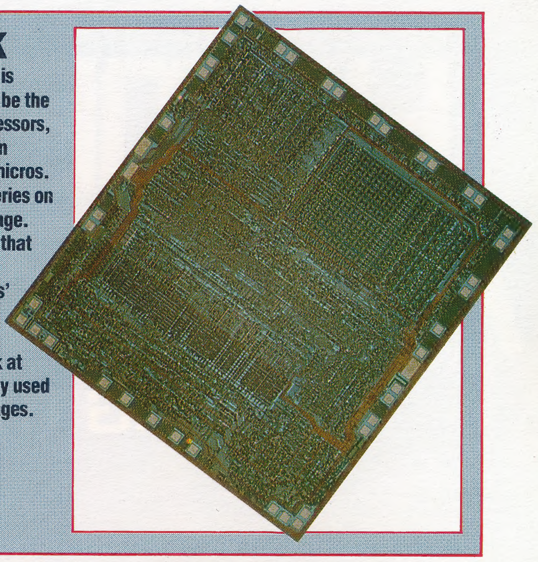
MACHINE CODE

CHANGE OF KEY Through the Amstrad CPC operating system, we can reconfigure the default values of the keyboard 1678

INDEX A complete index to issues 73 to 84 INSIDE BACK COVER

Next Week

- The Motorola 68000 is considered by many to be the best of the 16-bit processors, and is being used on an increasing number of micros. We now begin a new series on 68000 assembly language.
- Surveys have shown that word processing is the most common 'serious' application used by home micro owners. A new series takes a look at some of the most widely used word processing packages.



QUIZ

1) What function is performed by the data division of a COBOL program?
2) What are 'streams' and why are they necessary?
3) Why does object-oriented programming lend itself easily to the 'modular approach'?
4) What is a microprogrammable computer used for?

Answers To Last Week's Quiz

1) It changes any extra features written in extended FORTRANs into equivalents in standard FORTRAN.
2) Four seconds.
3) An operating system capable of simultaneously listening for incoming bytes and processing data already received.

Coming Up . . .

- Careers in computing are many and varied. We will be focusing on five of the major areas.
- Our Programming Project series will cover various aspects and usages of text compression
- A look at the Amstrad PCW 8256.

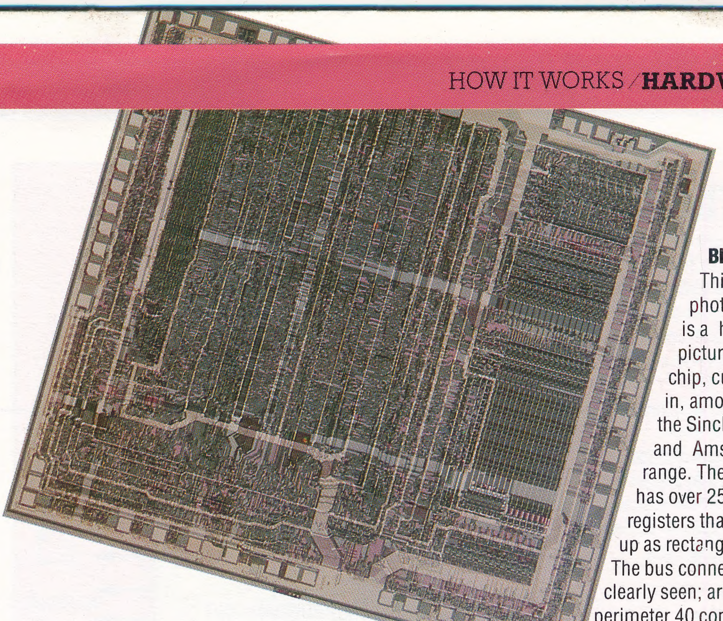
Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Deputy Editor** Steve Colwill; **Production Editors** Bobby Pickering, Jon Kaye; **Designers** Julian Dorr, Mike Clowes; **Staff Writer** Steve Malone; **Art Assistant** Caroline Clayton; **Sub Editor** Nik Lumsden; **Contributors** Chris Honey, Chris Laing, Dougie Bern, Richard Forsyth, Nigel Cross, Mike Curtis, Steve Cooke, Steve Colwill, Steve Malone, Martin Young; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Susan Brown; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
UK/EIRE – Issue Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Issue Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Issue Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Issue Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Issue Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Issue Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Issue Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Issue Price: Aus$2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Issue Price: NZ$2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited. Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY PAUL CHAVE

TOURING THE CIRCUIT

Understanding the complex interplay of the components that make up a microcomputer isn't easy. We begin a series of tours around the printed circuit boards of typical home micros, discussing the functions of the various components and how they interact. We start our guided tour at the most obvious point — the microprocessor.



Blow Up!
This photo-micrograph is a highly magnified picture of a Zilog Z80 chip, currently used in, among others, the Sinclair Spectrum and Amstrad CPC range. The Z80 has over 25 on-chip registers that show up as rectangular patterns. The bus connections can be clearly seen; around the perimeter 40 connective pads link the chip to the legs which allow it to be mounted in a PCB

Looking at a photomicrograph of a typical microprocessor, we can make out geometrical grids, which are the processor's internal registers, a number of rectangular pads around the outside to which external connections are made, and areas of jumbled logic circuitry.

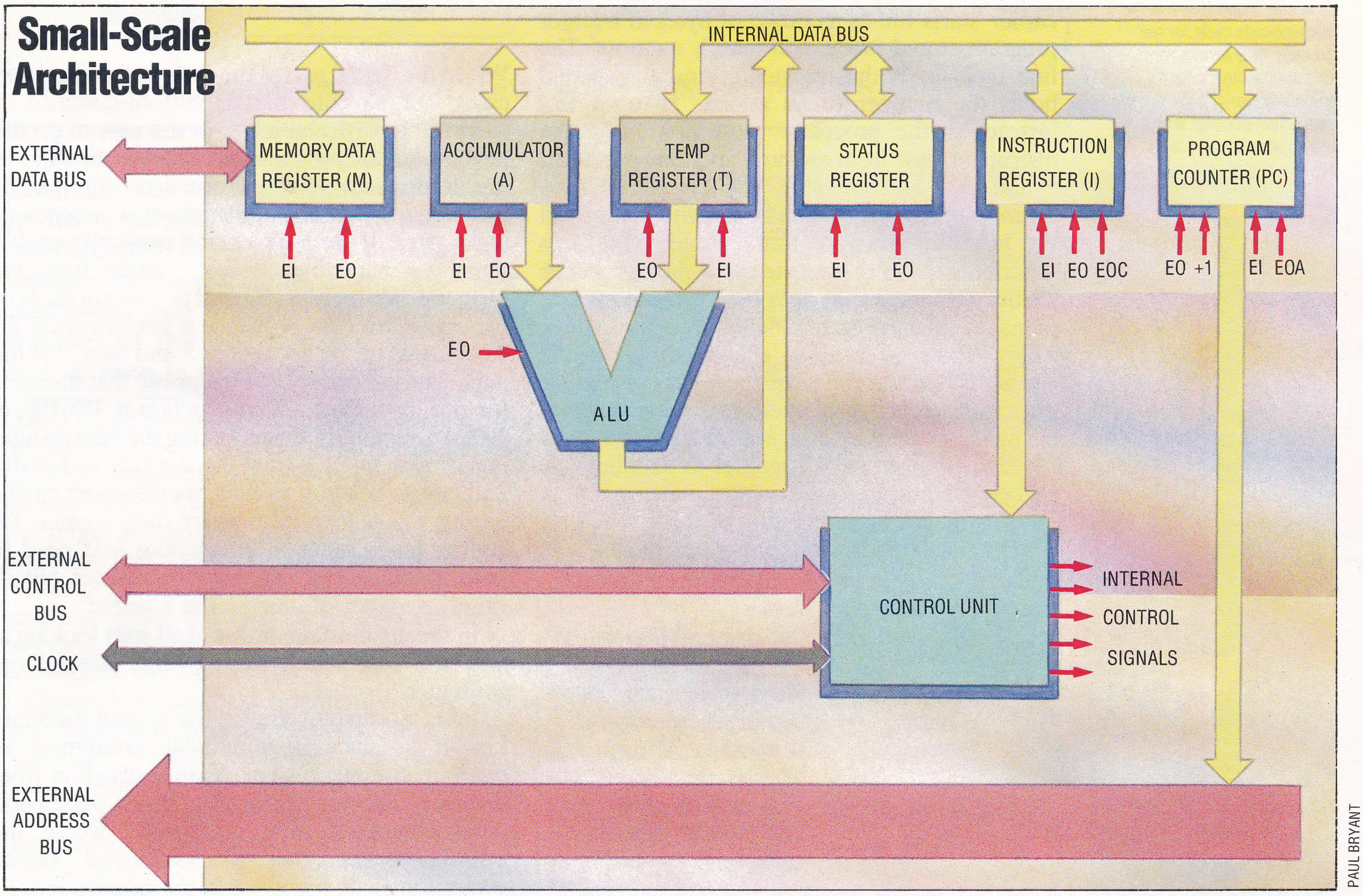
It is also possible to discern groups of lines that are the internal communication links between the various components on the chip. We can see that the architecture of a processor is determined by the constraints of two-dimensional layout and by the physical number of components that can be put on to a chip.

ARCHITECTURE

What processor instructions really do is set in motion a chain of electronic 'microevents' and this is the point at which the line dividing software and hardware is drawn. Let's illustrate this by taking the example of a simple instruction: adding a number to the value in the accumulator and putting the result back in the accumulator.

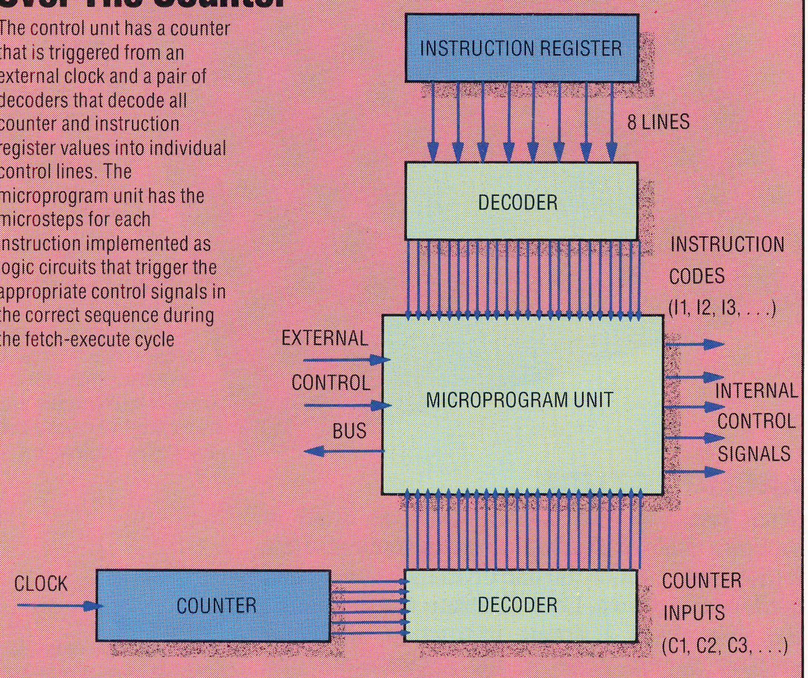
The diagram given here shows a typical example of processor architecture in a simplified form. All the internal registers are linked via an internal data bus and each register has control lines connected to it so that all data transfer operations

The Processor
The diagram here shows the typical architecture of an eight-bit processor. The on-chip components communicate via an internal data bus and internal control signals, but keep in touch with memory and other devices via control signals and external data and address buses



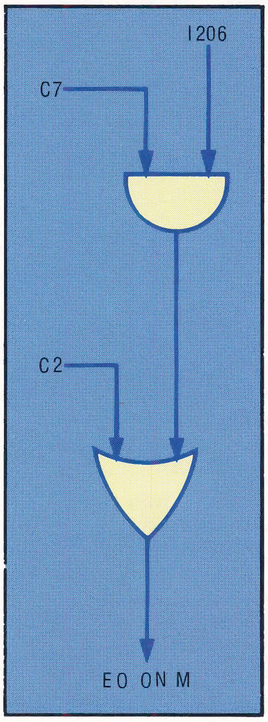
Over The Counter

The control unit has a counter that is triggered from an external clock and a pair of decoders that decode all counter and instruction register values into individual control lines. The microprogram unit has the microsteps for each instruction implemented as logic circuits that trigger the appropriate control signals in the correct sequence during the fetch-execute cycle



CAROLINE CLAYTON

Microstep Circuit
One particular control signal, output enable on the memory data register, needs to be set up at step 2 and step 7 of the fetch-execute cycle for our example ADD instruction. This simple logic circuit combines signal 206 from the instruction decoder with the two appropriate decoded counter signals to trigger EO on M



between registers and the data bus are synchronised. These control lines link into a central control unit, which will be discussed in more detail later.

The immediate ADD instruction mentioned earlier starts life as a couple of bytes in a larger program stored in the computer's memory. The first byte holds the instruction and the second holds the number to be added. To obey this instruction the processor must first bring the instruction byte from memory and then execute it — the fetch-execute cycle. The processor knows where to get the instruction from, as its address will be held in the program counter (PC). In describing the fetch-execute cycle, the part we are particularly interested in is how the control unit manages the various transfers between registers. This is done by sending 'enable' signals to the correct register at the right time. The system we have used to distinguish these signals is as follows:
EI enable input to register from internal data bus
EO enable output from a register to data bus
EOC enable output to the control unit
EOA enable output to the address bus

The 'fetch' cycle comprises these steps:

Step	Control Signals	Action
1	EOA on PC	Copy contents of program counter onto address bus
2	+1 on PC	Increment program counter
3	EO on M EI on I	Copy contents of memory data register into instruction register
4	EOC on I	Copy contents of instruction register into control unit

At the end of these four steps we've got the ADD instruction into the control unit and the program counter pointing at the byte holding the number to be added. Before looking at the way in which the instruction is obeyed let's look at the control unit.

The control unit comprises a register into which the current instruction is put, a decoder that decodes the 8 bits that make up the instruction byte into 256 separate lines, one for each possible value that the byte can hold. (If not all 256 possible codes are used for instruction codes the number of decoded lines may be less). These decoded lines feed into the microprogram unit, together with the decoded output from a counter. The microprogram unit is where the actual function of each instruction code is held as a logic circuit, and the outputs from these logic circuits form the various register enable signals. Let's take a look at the 'execute' part of the cycle:

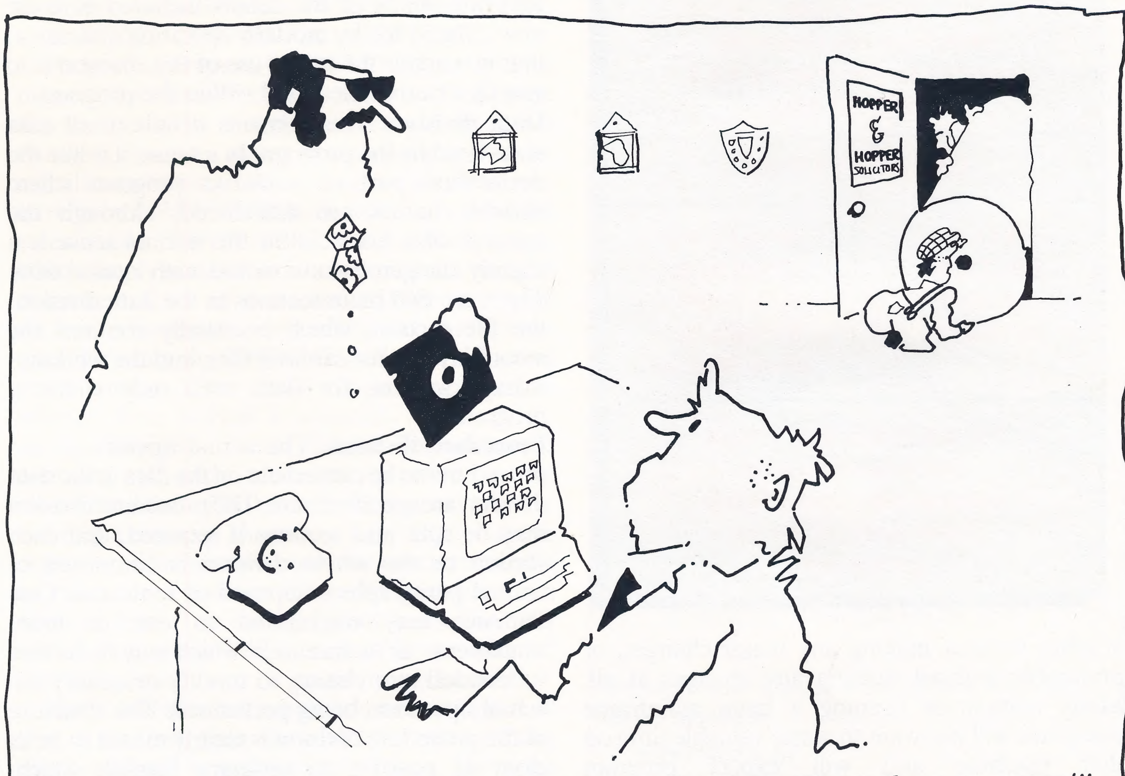
Step	Control Signals	Action
5	EOA on PC EI on M	Bring number to be added to M
6	+1 on PC	Increment program counter ready for next instruction
7	EO on M EI on T	Copy number into ALU temporary register
8	select ADD function EO on T EO on A	Perform addition
9	EO on ALU EI on A	Copy result into accumulator

Whilst the 'fetch' part of the cycle is the same in all cases, the 'execute' part is custom-designed to carry out a particular job — in this case to get the next byte in the program and add it to the value in the accumulator. Each instruction code has its own microprogram implemented as a series of logic gates. If we look at one particular enable signal — that which enables output from the memory data register (EO on M) — we can see how the above 'execute' sequence is implemented. EO on M needs to be set at step 3 and step 7 of the fetch-execute cycle. Let us suppose that the code for our immediate ADD instruction is 11001110, or 206 in decimal. By simply gating the step counter line C7 with line 206 from the instruction decoder and ORing the result with C3, we obtain an enable output signal for the memory data register. Of course, many different instructions may need to send enable output signals to this register, but it's simply a matter of ORing these together.

In the next instalment we shall start looking at how the complete system is built around the computer.

Processor manufacturers use special machines known as microprogrammable computers to program the micro-steps of an instruction from software. These are often used in the design and development of new processors, where the instruction set can be developed before committing it to hardware in the control unit.

COMMON USAGE



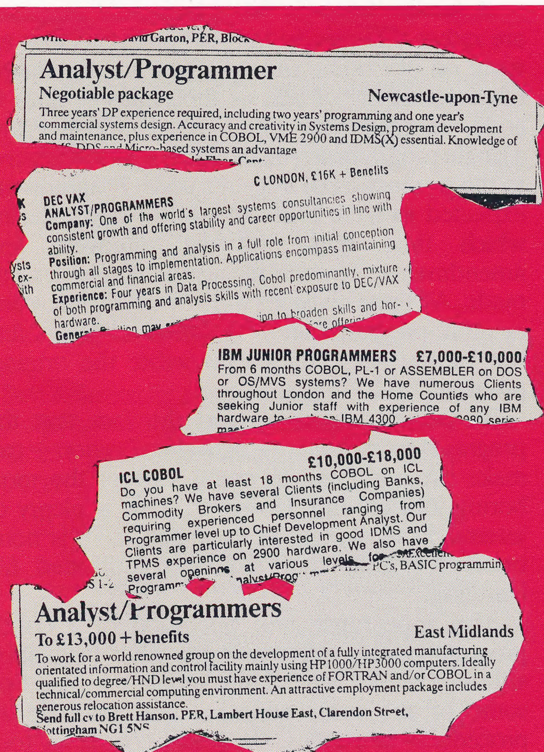
'HERE'S YOUR COBOL SOFTWARE SIR...OH, AND YOU'LL BE NEEDING SOME PAPER FOR THE LISTINGS!!!

MIKE CLOWES

Although little use has been made of COBOL on home micros, its wide use and easy transferability have made it a leader for commercial applications. In this first instalment, we outline the four main divisions of the language and discuss the data division in more detail.

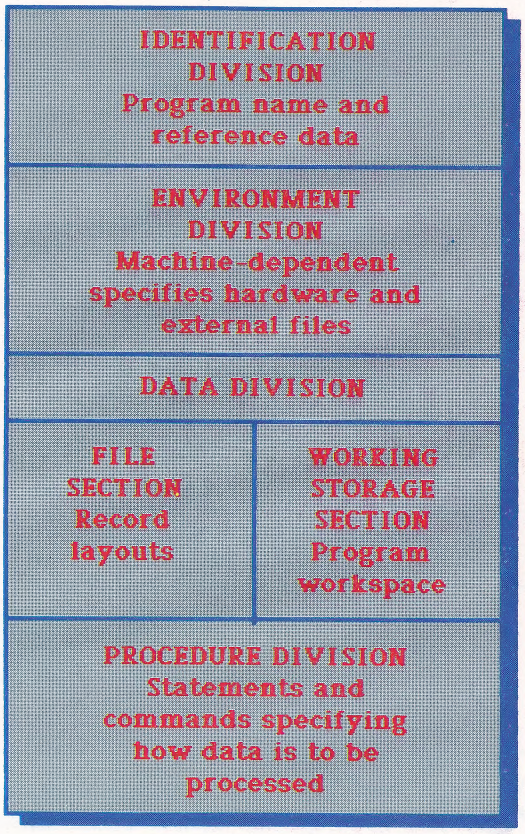
COBOL (Common Business Oriented Language) is by far the most widely used programming language worldwide, and although BASIC may be the more widely known, COBOL probably accounts for more actual lines of code than all other languages put together. This isn't just because COBOL is verbose, needing many lines for even a simple program, but because it's been the major language (almost the only language) for commercial and business use.

COBOL is a heavily standardised language. In fact, there is a permanent body, the CODASYL committee, which oversees its use and development. This high degree of standardisation is important for several reasons. It's vital that a commercial organisation should be able to transfer its software, which could easily be one of its most valuable assets, from one computer to



Program For Profit

Despite their age, both COBOL and FORTRAN are still widely used by businesses across the world. Courses in these and other languages are often available locally, and can lead to the prospect of secure employment and good salaries



CAROLINE CLAYTON

Divided Structure
Programmers familiar only with BASIC may find the structure of a COBOL program somewhat alien, but it is well-adapted to the demands made on the language, especially readability, portability, and the need for frequent checking and updating

another without making any major changes, or preferably without making any changes at all. Many companies running a large mainframe computer will not want to waste valuable time on that machine and will expect program development to be done on a completely different machine, so again it's vital that the programs can be transferred easily.

A further factor is that commercial programs are rather different from BASIC programs that we write on our home micros. They are very large, often consisting of tens of thousands of lines of code, and tend to be written by a team rather than a single programmer, all of which is not conducive to the production of bug-free code. Programs are expected to have a life of several years, during which time bugs will be removed, and amendments, extensions and improvements made. All these revisions will be carried out by one or more programmers who will in all probability not be the same ones who wrote the original program.

These restrictions — and the fact that the programs are written to a rigid specification — can make programming in COBOL one of the most boring jobs around. However, there is plenty of interest in the language itself and there is none better for tasks that require complicated file handling.

The appearance and operation of a COBOL program is very different from other languages, although buried beneath the verbiage we can find some of the same concepts that underlie all procedural languages. Programs consist of four divisions:

Identification division. This is mainly documentary, and provides a place for a program name, programmer's name, dates written and compiled and introductory remarks.

Environment division. This is intended to be the machine-dependent part, containing details of the computer on which the program was written and run. Many of the details included here are now catered for by modern operating systems so that nowadays the major use of this division is to specify external files used within the program.

Data division. This contains details of all data areas used by the program. In a sense, it is like the declaration part of a PASCAL program where variable names are introduced, although the concept of a 'variable' in the normal sense is a slightly dangerous one to use with COBOL data. There are two main sections in the data division: the file section, which essentially contains the record layouts for external files, and the working-storage section, for data used only within a program.

Procedure division. The actual operations and procedures to be carried out on the data in the data division are specified here. The procedure division may be split into sections if required, and each section or the whole division is composed of named paragraphs composed of sentences. One sentence may correspond to one or more 'statements' or 'commands' which may be further subdivided into clauses to modify or qualify the actual operation being performed. The structure of the procedure division is clearly meant to be as close as possible to ordinary English which, coupled with long identifiers of up to thirty characters, can lead to programs that are understandable not only to other programmers but to non-programmers as well. However, it is just as easy to write incomprehensible 'spaghetti' code in COBOL as in any other language.

We'll concentrate here on the data division, since defining and naming the data areas is a very important part of a COBOL program. The COBOL view of data is to regard the entire data division as one long string of characters that can be divided and subdivided at will. It is best to think of the data division as being simply a block of memory. The various divisions made in this data block are all named and given a level number to indicate the basic structure of the data.

Normal level numbers range from 01 to 49, with some additional ones. A section of the data division will be defined at level 01. This section can then be subdivided by giving higher level numbers to the subsections — 02 for example — though the numbers do not have to be consecutive. One of these subsections can be further subdivided by using an even higher level number, and so on.

Data names can be up to 30 characters long, using only alphabetic and numeric characters and hyphens. COBOL has a very large vocabulary of reserved words that cannot be used for data names. However, few of the reserved words have

hyphens so this problem is not too difficult to overcome.

We need to distinguish between two types of data items: elementary and group. An elementary item is one that is not further subdivided; both elementary and group items can be referred to in the procedure division. Each elementary data item must have a USAGE and a PICTURE. There are two main USAGEs: COMPUTATION and DISPLAY.

As you may expect, COMPUTATIONAL is used for numeric data items for calculations and will cause the number to be stored in a binary form rather than as a character string, thus making arithmetic calculations faster. A data item does not, however, have to be COMPUTATIONAL in order to be used for calculations. DISPLAY is the normal default usage, which simply means the data is stored as a string of characters. Each of these two USAGEs may have further refinements, depending on the system used; for example, COMPUTATIONAL-3 for a floating point number. The USAGE clause is optional; if omitted, then DISPLAY is assumed.

The PICTURE clause defines the expected contents of each character position in an elementary item. The main symbols used are: 9 to indicate a numeric character; A for an alphabetic character; X for an alphanumeric character; and V to indicate the position of the decimal point in a numeric item. The decimal point is not normally stored, but COBOL keeps track of its position so that calculations can be properly performed.

An S indicates that a numeric item is signed. If this is omitted, then the absolute value of any number stored is automatically taken. There are options as to whether the sign is stored as a separate character or not. (Note that PIC is short for PICTURE, and either can be used.) Finally, as well as a PICTURE and a USAGE, it is possible to assign an initial value to any data item, or to specify it as an array.

PROGRAM LAYOUT

Like FORTRAN, COBOL was originally designed for use with punched cards. Each line should not exceed 72 characters in length and, if necessary, a continuation line must be used. The layout is strictly specified, but many systems will allow a more free-form layout if the program is not destined for another machine. The first six columns are used for line numbers. COBOL does not require line numbers but they may be put in to help with debugging. Column 7 is used to mark a continuation line, or an asterisk in this position indicates a comment. Columns 8 to 11 are the A area. Division, section and paragraph names begin here as do 01 level data items. Other data items and statements in the procedure division begin in the B area — columns 12 to 72. Most data declarations constitute a sentence and so they must end with a full stop. If you need characters in a data item that are not actually referred to in the procedure division, then the reserved word FILLER is used.

In the next part of this series, we'll look at the procedure division.

Personal Details

Data definition for a record on a personnel file:

```
01 EMPLOYEE-RECORD.
   02 EMPLOYEE-NAME.
      03 EMPLOYEE-INITIAL PIC A.
      03 FILLER PIC X.
      03 EMPLOYEE-SURNAME PIC X(15).
   02 EMPLOYEE-DEPARTMENT PIC XXX.
   02 EMPLOYEE-SALARY-SCALE PIC 9.
   02 EMPLOYEE-DOB.
      03 DOB-DAY PIC 99.
      03 DOB-MONTH PIC 99.
      03 DOB-YEAR PIC 99.
```

Personnel Record
The data types used to maintain a record of personal information about employees are shown. Note the use of PIC to specify expected data types — 9 indicates a number, A an alphabetic character, and X an alphanumeric character

Data definition for a line of headings to produce a report from the above file:

```
01 HEADING-LINE.
   02 HEADING-1 PIC X(4) VALUE 'NAME'.
   02 FILLER PIC X (20) VALUE SPACES.
   02 HEADING-2 PIC X(10) VALUE
      'DEPARTMENT'.
   02 FILLER PIC X(3) VALUE SPACES.
   02 HEADING-3 PIC X(9) VALUE 'SAL SCALE'.
   02 FILLER PIC X(10) VALUE SPACES.
   02 HEADING-4 PIC X(3) VALUE 'DOB'.
```

Layout Design
The layout of the report to be generated is defined here. Data items can have initial values assigned to them. HEADING-2, for example, consists of 10 alphanumeric characters and has been assigned the value DEPARTMENT

Data definition for a line of output for a report from the personnel file:

```
01 OUTPUT-LINE.
   02 OUTPUT-NAME
      03 OUTPUT-INITIAL PIC A.
      03 FILLER PIC X VALUE'.'.
      03 OUTPUT-SURNAME PIC X(15).
   02 FILLER PIC X(7) VALUE SPACES.
   02 OUTPUT-DEPARTMENT PIC XXX.
   02 FILLER PIC X(10) VALUE SPACES.
   02 OUTPUT-SALARY-SCALE PIC 9.
   02 FILLER PIC X(18) VALUE SPACES.
   02 OUTPUT-DOB.
      03 OUTPUT-DAY PIC 99.
      03 FILLER PIC X VALUE'/'
      03 OUTPUT-MONTH PIC 99.
      03 FILLER PIC X VALUE'/'
      03 OUTPUT-YEAR PIC 99.
```

Filling Space
These lines specify the line of output from the personnel file. Characters can be included in data items even if not referred to by the procedure division — this is done using the reserved word FILLER, which is used here and in the previous listing to insert spaces

Data definition for a one-dimensional array of twenty 3-digit numbers and a two-dimensional array of 10*20 3-digit numbers (signed — with one decimal place):

```
01 DATA-TABLE-1.
   02 DATA-TABLE-ENTRY PIC S99V9 USAGE
      COMPUTATIONAL OCCURS 20 TIMES.
01 DATA-TABLE-2.
   02 DATA-TABLE-ROW OCCURS 10 TIMES.
      03 DATA-TABLE-ENTRY-2 PIC S99V9
         USAGE
         COMPUTATIONAL OCCURS 20 TIMES.
```

Built For Speed
Declaring data as COMPUTATIONAL, as this section of program does, means that the values are stored in binary format rather than as characters. This makes any subsequent calculations that use the data much faster. However data does not necessarily have to be defined in this way if you want to do arithmetic operations on it, but calculations on data held as characters will be slower

SPREADING OUT

Having covered the basic functions that allow us to enter data and formulae into our spreadsheet, we turn our attention to additional features that make the spreadsheet a useful financial modelling tool.

When setting up a spreadsheet to make a series of calculations, the formulae to be entered into each cell of a particular column or row are often similar. For example, we may wish to add each element in the first column to each element in the second column and put the results in the third column. To perform this simple task a formula would need to be entered into each cell in the third column. The first cell in this column, A3, would need to be programmed with the formula A1+A2; the next cell in the third column, B3, would need to be programmed with the formula B1+B2, and so on. Although the formulae are not identical they are similar, so to avoid the tedious task of having to enter the formulae manually into every cell in the third column, it would be useful to have a facility that automatically entered suitably amended formulae into a complete (or part) row or column. This is the purpose of the Replicate function.

Clear, Store and Retrieve functions act on the complete sheet and, as the names imply, allow the user to clear the cell data from the complete sheet, store the data for the complete sheet in memory (rather than on disk or tape) and recall it later.

Another useful function to assist in moving around the sheet is the Tab function. Although it is possible to move to any part of the sheet using the cursor, this can be rather slow, particularly as the sheet has to be rewritten each time the screen window scrolls across the sheet. The Tab function allows the user to jump to any cell simply by entering the cell name.

THE REPLICATE FUNCTION

The subroutines that handle the Replicate function are located between lines 5400 and 5995. The first routine takes the formula to be replicated and splits it up into its constituent elements. This routine is in fact called by the main Replicate routine and works on a formula taken from the infix formula array F$() and stored in C$. The formula is broken down into its operators, cell names and constants and placed in the array E$(). The routine scans through C$ a character at a time, and if the current character is not an operator it is added to a temporary string, T$. If the character *is* an operator then T$ is placed in the next free element of E$() (as encountering an operator means that a complete operand is stored in T$). The operator just encountered is also added to E$(). The process then continues until the entire formula has been scanned.

The next section of code, between lines 5500 and 5650, contains the input and verify routine. This allows the user to direct the replicate function to act on a particular group of cells and is written to accept commands in the format A1(B1-F1. This means: take the formula in cell A1 and replicate it in column 1 between B1 and F1. Alternatively, to replicate across a row, a command like this might be entered: A1(A2-A10. The first section of the input routine, between lines 5500 and 5520, checks that the input has the correct syntax. Next, the input is divided to extract the three cell names that need to be worked on. These are then placed in R1$, R2$ and R3$.

The next section, at line 5700, is the actual entry point to the Replicate routine, its function being to decide whether the command implies that a row or a column is to be replicated and jump to the appropriate replicate row or column routine. It is possible to decide if a row or column is to be replicated by examining the three cell names input as part of the replicate command. Each cell name is made up of a letter followed by a number. If each of the three cell names starts with the same letter, as in A1(A2-A10, then it is clear that the formula replication should take place across a row, in this case row A. If, however, the number part in each cell name is the same, for example A3(B3-H3, then a column is to be replicated. One spin-off of checking for row or column is that illegal entries such as A1(B1-F2 are automatically picked up.

At lines 5800 to 5895 we find the code that replicates a formula down a column. First, the routine we discussed earlier that splits up the formula to replicate is called. A FOR...NEXT loop at line 5820 uses the ASCII values of the first characters of R2$ and R3$ (the cell names between which the formula is to be replicated) as its control variables. The task of constructing the new formula for each cell is then started at line 5830.

At this stage, it is worth discussing the limitations of the Replicate function. When you replicate a formula down a column, the function will only work successfully if each cell name within the formula to be replicated has the same row letter. This means that a formula such as A1+A2*A3 will replicate successfully but A1+A2*B2 will not. This is not a serious restriction, however, as most applications that make use of the Replicate function use formulae that work on elements within a single row.

THE CLEAR, STORE, RETRIEVE AND TAB FUNCTIONS

The code for all these short routines is given between lines 5000 and 5395. Clearing the sheet is a simple matter. The array M(,), used to hold the cell data for the complete sheet is simply set to zero using a couple of nested FOR...NEXT loops, and the sheet is reprinted to the screen by the subroutine

call to line 1700.

To store and retrieve a sheet of data, a second array, N(,), is used. The store and retrieve routines simply transfer the contents of M(,) to N(,), or vice versa.

The Tab routine starts at line 5200 and accepts a cell name as input. Before rewriting the sheet, the horizontal and vertical limits of the screen window need to be recalculated. In addition to reprinting the sheet data, new column and row label have to be printed to take account of the fact that the window on to the sheet may have to be changed as a consequence of moving to the new cell.

In the next, and final , instalment of this project, we shall look at methods of saving and loading spreadsheet data and formulae to disk or tape.

Basic Flavours

BBC Micro:

Make the following changes to the Commodore 64 version (ensure that the PRINT statements at lines 5220 and 5502 contain enough spaces to erase a complete screen line):

```
5210 PRINT TAB (0,22);
5220 PRINT "NEW CELL:      "
5501 PRINT TAB (0,22);
5502 PRINT "REPLICATE:     "
```

Amstrad CPC 464/664

Make the following changes to the Commodore 64 version (ensure that the PRINT statements at line 5220 and 5502 contain enough space to erase a complete line):

```
5210 LOCATE 1,22
5220 PRINT "NEW CELL:      "
5501 LOCATE 1,22
5502 PRINT "REPLICATE:     "
```

Additional Features

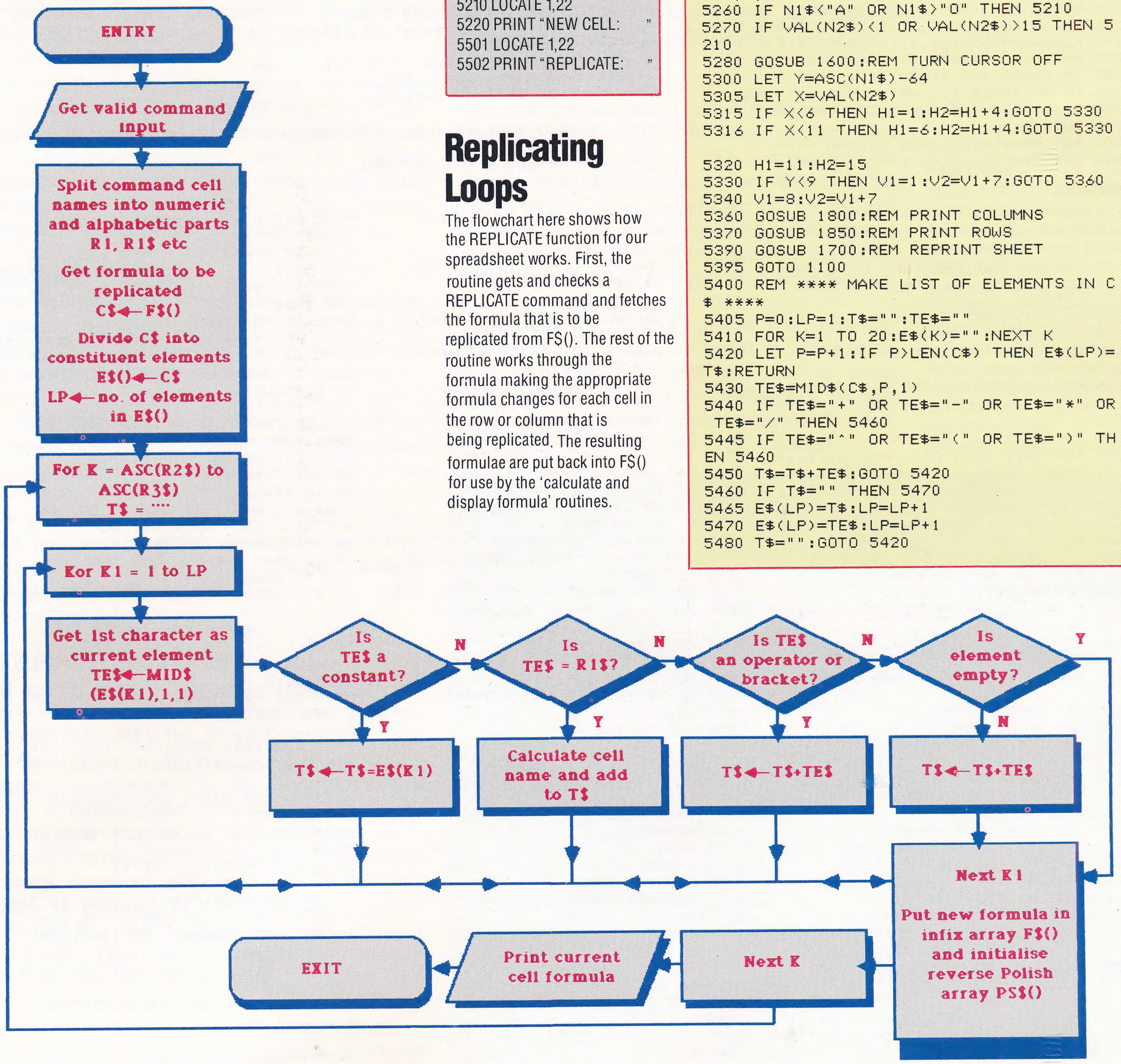
Commodore 64:

```
2320>LET P$=H$((J-1)*15+I,1TO ):
LET I$=F$((J-1)*15+I,1TO )
5000 REM **** CLEAR ARRAY ****
5010 FOR I=1 TO 15:FOR J=1 TO 15:M(I,J)=
0:NEXT J,I:GOSUB 1700:RETURN
5100 REM **** GET PREVIOUS SHEET ****
5110 FOR I=1 TO 15:FOR J=1 TO 15
5120 M(I,J)=N(I,J):NEXT J,I
5130 GOSUB 1700:RETURN:REM PRINT DATA
5150 REM ***** STORE CURRENT SHEET ****
5160 FOR I=1 TO 15:FOR J=1 TO 15
5170 N(I,J)=M(I,J):NEXT J,I
5180 GOSUB 1700:RETURN:REM PRINT DATA
5200 REM **** GOTO CELL ROUTINE ****
5210 GOSUB 1950:REM MOVE TO INPUT LINE
5220 PRINT "NEW CELL:
      ";CU$
5230 INPUT "NEW CELL:";NC$
5240 LET NC$=MID$(NC$,1,3)
5250 LET N1$=MID$(NC$,1,1):LET N2$=MID$(
NC$,2,2)
5260 IF N1$<"A" OR N1$>"O" THEN 5210
5270 IF VAL(N2$)<1 OR VAL(N2$)>15 THEN 5
210
5280 GOSUB 1600:REM TURN CURSOR OFF
5300 LET Y=ASC(N1$)-64
5305 LET X=VAL(N2$)
5315 IF X<6 THEN H1=1:H2=H1+4:GOTO 5330
5316 IF X<11 THEN H1=6:H2=H1+4:GOTO 5330

5320 H1=11:H2=15
5330 IF Y<9 THEN V1=1:V2=V1+7:GOTO 5360
5340 V1=8:V2=V1+7
5360 GOSUB 1800:REM PRINT COLUMNS
5370 GOSUB 1850:REM PRINT ROWS
5390 GOSUB 1700:REM REPRINT SHEET
5395 GOTO 1100
5400 REM **** MAKE LIST OF ELEMENTS IN C
$ ****
5405 P=0:LP=1:T$="":TE$=""
5410 FOR K=1 TO 20:E$(K)="":NEXT K
5420 LET P=P+1:IF P>LEN(C$) THEN E$(LP)=
T$:RETURN
5430 TE$=MID$(C$,P,1)
5440 IF TE$="+" OR TE$="-" OR TE$="*" OR
 TE$="/" THEN 5460
5445 IF TE$="^" OR TE$="(" OR TE$=")" TH
EN 5460
5450 T$=T$+TE$:GOTO 5420
5460 IF T$="" THEN 5470
5465 E$(LP)=T$:LP=LP+1
5470 E$(LP)=TE$:LP=LP+1
5480 T$="":GOTO 5420
```

Replicating Loops

The flowchart here shows how the REPLICATE function for our spreadsheet works. First, the routine gets and checks a REPLICATE command and fetches the formula that is to be replicated from F$(). The rest of the routine works through the formula making the appropriate formula changes for each cell in the row or column that is being replicated. The resulting formulae are put back into F$() for use by the 'calculate and display formula' routines.



CAROLINE CLAYTON

```
5500 REM ***  REPLICATING FORMULA *****
5501 GOSUB 1950
5502 PRINT "REPLICATE:
        ";CU$
5503 INPUT "REPLICATE:";R$
5505 LET P=0:R1$="":R2$="":R3$="":T$=""
5510 IF MID$(R$,3,1)<>"(" AND MID$(R$,4,
1)<>"(" THEN 5500
5515 IF MID$(R$,6,1)<>"-" AND MID$(R$,7,
1)<>"-" AND MID$(R$,8,1)<>"-" THEN 5500
5517 IF R$="" THEN 5500
5520 LET P=P+1
5530 LET T$=MID$(R$,P,1)
5540 IF T$="(" THEN P=P+1:GOTO 5570
5550 LET R1$=R1$+T$
5560 GOTO 5520
5570 LET T$=MID$(R$,P,1)
5580 IF T$="-" THEN P=P+1:GOTO 5610
5590 LET R2$=R2$+T$
5600 LET P=P+1:GOTO 5570
5610 LET T$=MID$(R$,P,1)
5620 LET R3$=R3$+T$
5630 LET P=P+1
5640 IF P<=LEN(R$) THEN 5610
5650 RETURN
5700 REM **  DECIDE COLUMN OR ROW ***
5730 GOSUB 5500:REM TEST FOR VALID INPUT
5765 R1=VAL(MID$(R1$,2)):R1$=MID$(R1$,1,1)
5770 R2=VAL(MID$(R2$,2)):R2$=MID$(R2$,1,1)
5775 R3=VAL(MID$(R3$,2)):R3$=MID$(R3$,1,1)
5780 IF R1=R2 AND R1=R3 THEN 5800
5790 IF MID$(R1$,1,1)=MID$(R2$,1,1) AND
MID$(R1$,1,1)=MID$(R3$,1,1) THEN 5900
5795 GOTO 5700
5800 REM **** REPLICATE COLUMN ****
5805 LET C$=F$((ASC(R1$)-65)*15+R1)
5810 GOSUB 5400:REM MAKE LIST OF ELEMENTS
5820 FOR K=ASC(R2$) TO ASC(R3$):T$=""
5830 FOR K1=1 TO LP
5840 LET TE$=MID$(E$(K1),1,1)
5845 IF (TE$>="0" AND TE$<="9") OR TE$="
." THEN T$=T$+E$(K1):GOTO 5880
5850 IF TE$=R1$ THEN T$=T$+CHR$(K)+MID$(
E$(K1),2):GOTO 5880
5860 IF TE$="+" OR TE$="-" OR TE$="*" TH
EN T$=T$+TE$:GOTO 5880
5865 IF TE$="/" OR TE$="^" OR TE$="(" OR
 TE$=")" THEN T$=T$+TE$:GOTO 5880
5870 IF TE$<>""THEN T$=T$+TE$
5880 NEXT K1
5890 F$((K-65)*15+R1)=T$:PS$((K-65)*15+R
1)=""
5895 NEXT K:GOSUB 1900:RETURN
5900 REM **** REPLICATE ROW ****
5905 LET C$=F$((ASC(R1$)-65)*15+R1)
5910 GOSUB 5400:REM MAKE LIST OF ELEMENTS
5920 FOR K=R2 TO R3:T$=""
5930 FOR K1=1 TO LP
5940 LET TE$=MID$(E$(K1),1,1)
5945 IF (TE$>="0" AND TE$<="9") OR TE$="
." THEN T$=T$+E$(K1):GOTO 5980
5950 IF TE$>="A" AND TE$<="O" THEN T$=T$
+TE$+MID$(STR$(K),2):GOTO 5980
5960 IF TE$="+" OR TE$="-" OR TE$="*" TH
EN T$=T$+TE$:GOTO 5980
5965 IF TE$="/" OR TE$="^" OR TE$="(" OR
 TE$=")" THEN T$=T$+TE$:GOTO 5980
5970 IF TE$<>""THEN T$=T$+TE$
5980 NEXT K1
5990 F$((ASC(R1$)-65)*15+K)=T$:PS$((ASC(
R1$)-65)*15+K)=""
5995 NEXT K:GOSUB 1900:RETURN
```

Spectrum:

```
5000>REM *** CLEAR ARRAY ****
5010 DIM M(15,15): GO SUB 1700:
RETURN
5100 REM *** GET PREVIOUS SHEET****
5110 FOR I=1 TO 15: FOR J=1 TO 15
5120 LET M(I,J)=N(I,J)
5130 NEXT J: NEXT I
5140 GO SUB 1700: RETURN
5150 REM STORE CURRENT SHEET IN
MEMORY
5160 FOR I=1 TO 15: FOR J=1 TO 15
5170 LET N(I,J)=M(I,J): NEXT J:
NEXT I
5180 GO SUB 1700: RETURN
5200 REM **** GOTO CELL ROUTINE****
5210 PRINT AT 18,0;"        ENTER
NEW CELL          "
5220 INPUT  LINE N$
5225 IF LEN (N$)<3 THEN  LET N$=
N$+" "
5230 LET N$=N$(1 TO 3): LET O$=N
$(2 TO 3)
5240 LET N$=N$(1)
5250 IF N$<"A" OR N$>"O" THEN  G
O TO 5210
5260 IF VAL (O$)<1 OR VAL (O$)>1
5 THEN  GO TO 5210
5280 GO SUB 1600: REM TURN CURSO
R OFF
5300 LET Y=CODE (N$)-64
5305 LET X=VAL (O$)
5310 LET VL=V2: LET HL=H2
5315 IF X<H1 THEN  GO TO 5330
5316 IF X<H2 THEN  GO TO 5335
5320 IF X>=12 THEN  LET H1=12: L
ET H2=15: GO TO 5340
5330 LET H1=X: LET H2=H1+3
5335 IF Y<V1 THEN  GO TO 5350
5336 IF Y<V2 THEN  GO TO 5360
5340 IF Y>=9 THEN  LET V1=9: LET
 V2=V1+6: GO TO 5360
5350 LET V1=Y: LET V2=Y+6
5360 GO SUB 1800: REM PRINT COLUMNS
5370 GO SUB 1850: REM PRINT ROWS
5380 IF V2=VL AND H2=HL THEN  GO
 SUB 1650: GO TO 1100
5390 GO SUB 1700
5395 GO TO 1100
5400 REM **** MAKE LIST OF ELEME
NTS IN I$ ***
5405 LET P=0: LET LP=1: LET T$="
": LET U$=""
5410 DIM E$(20,5)
5420 LET P=P+1: IF P>LEN (C$) TH
EN  LET E$(LP)=T$: RETURN
5430 LET U$=C$(P)
5440 IF U$="+" OR U$="-" OR U$="
*" OR U$="/" THEN  GO TO 5460
5445 IF U$="^" OR U$="(" OR U$="
)" THEN  GO TO 5460
5450 LET T$=T$+U$: GO TO 5420
5460 LET E$(LP)=T$: LET LP=LP+1
5470 LET E$(LP)=U$: LET LP=LP+1
5480 LET T$="": GO TO 5420
5500 REM *** REPLICATING ****
5505 LET P=0: LET X$="": LET Y$=
"": LET Z$="": LET T$=""
5510 IF R$(3)<>"(" AND R$(4)<>"(
" THEN  GO TO 5660
5515 IF R$(6)<>"-" AND R$(7)<>"-
" AND R$(8)<>"-" THEN  GO TO 5660
5520 LET P=P+1
5530 LET T$=R$(P)
5540 IF T$="(" THEN  LET P=P+1:
GO TO 5570
5550 LET X$=X$+T$
5560 GO TO 5520
5570 LET T$=R$(P)
5580 IF T$="-" THEN  LET P=P+1:
GO TO 5610
5590 LET Y$=Y$+T$
5600 LET P=P+1: GO TO 5570
5610 LET T$=R$(P)
5620 LET Z$=Z$+T$
5630 LET P=P+1
5640 IF P<=LEN (R$) THEN  GO TO 5610
5650 RETURN
5670 PRINT AT 18,0;"ERROR IN REP
LICATE STRING"
5680 RETURN
5700 REM **** DECIDE COLUMN OR R
OW ****
5710 PRINT AT 18,0;"REPLICATE:
                  "
5720 INPUT  LINE R$
5730 GO SUB 5500
5765 LET R1=VAL (X$(2 TO )): LET
 X$=X$(1)
5770 LET R2=VAL (Y$(2 TO )): LET
 Y$=Y$(1)
5775 LET R3=VAL (Z$(2 TO )): LET
 Z$=Z$(1)
5780 IF R1=R2 AND R1=R3 THEN  GO
 TO 5800
5790 IF X$=Y$ AND X$=Z$ THEN  GO
 TO 5900
5795 GO TO 5700
5800 REM **** REPLICATE COLUMN *****
5805 LET C$=F$((CODE (X$)-65)*15+R1)
5810 GO SUB 5400: REM MAKE LIST
OF ELEMENTS
5820 FOR K=CODE (Y$) TO CODE (Z$
): LET T$=""
5830 FOR J=1 TO LP
5840 LET U$=E$(J)( TO 1)
5850 IF U$=X$ THEN  LET T$=T$+CH
R$ (K)+E$(J,2 TO ): GO TO 5880
5860 IF U$="+" OR U$="-" OR U$="
*" THEN  LET T$=T$+U$: GO TO 588
5865 IF U$="/" OR U$="^" OR U$=" 0
(" OR U$=")" THEN  LET T$=T$+U$:
 GO TO 5880
5870 IF U$<>"" THEN  LET T$=T$+U$
5880 NEXT J
5882 LET U$="": FOR i=1 TO LEN (
T$): IF T$(i TO i)<>" " THEN  LE
T U$=U$+T$(i TO i)
5885 NEXT i
5890 LET F$((K-65)*15+R1)=U$: LE
T H$((K-65)*15+R1)=""
5895 NEXT K: GO SUB 1900: RETURN
5900 REM **** REPLICATE ROW ****
5905 LET C$=F$((CODE (X$)-65)*15
+R1)
5910 GO SUB 5400: REM MAKE LIST
OF ELEMENTS
5920 FOR K=R2 TO R3: LET T$=""
5930 FOR J=1 TO LP
5940 LET U$=E$(J,1 TO 1)
5950 IF U$>="A" AND U$<="O" THEN
  LET T$=T$+U$+STR$ (K): GO TO 5980
5960 IF U$="/" OR U$="^" OR U$="
(" OR U$=")" THEN  LET T$=T$+U$:
 GO TO 5980
5970 IF U$<>"" THEN  LET T$=T$+U$
5980 NEXT J
5990 LET F$((CODE (X$)-65)*15+K)
=T$: LET H$((CODE (X$)-65)*15+K)
=""
5995 NEXT K: GO SUB 1900: RETURN
```

STANDARDISATION

While most consumers would prefer a level of standardisation among manufacturers, it has hardly been attained in practice. Ideally, all manufacturers would agree on specifications for products to remove the problem of incompatibility. In this way, not only would communication with a given device be standardised, but development costs would be reduced.

Unfortunately, manufacturers often have their own interpretation of a standard, which creates a market filled with incompatible products. An obvious example is the RS232C standard (see page 1528), which many companies have defined in different ways. This is particularly noticeable with regard to the wiring of the port, though a little soldering can usually overcome the problem. But some manufacturers have changed the voltage of the port so that it won't communicate directly with other RS232C devices at all, requiring the user to buy a new adaptor.

Two different approaches to standardisation can be seen in the MSX range of micros (see page 141) and in the CP/M operating system (see our series beginning on page 1264). The former depends on using the same hardware specifications, while the latter, on disk, provides compatibility for widely different machines.

STAR NETWORK

A *star network* is one of the several ways to arrange a local area network (LAN; see page 969). In this system, the computers constituting the nodes of the network are configured into a 'star', with a number of computers connected to a central machine that acts as a switching device for messages between the nodes. While this system does not require the computers on the edge of the network to have a switching mechanism of their own, it means that the computer in the central node is dedicated almost entirely to controlling communications. Furthermore, should a fault develop within the central computer, the entire network will break down.

STEPPER MOTORS

The *stepper motor* is one of the types of electric motor widely used in small-scale robotics projects. Its major attribute is that it can be turned through an exact angle in response to a digital pulse. Stepper motors work by having a large number of small electromagnetic coils surrounding the rotating shaft of the motor. These coils are affixed alternately to two different terminals so that when a charge is sent through one coil, an opposite charge is induced in the opposite terminal, deflecting the rotor to align with the north and south poles of the coils. By energising coil pairs in order, the motor can be made to rotate. The switching logic needed for this is often held on a 'driver' chip that accepts pulse and direction signals from the controlling device and outputs the appropriate signals to the stepper motor coils.

STORAGE DEVICE

Any device that can store information that can subsequently be retrieved for use by a computer's central processing unit is a *storage device*. These can range from registers and dynamic RAM chips to floppy disk drives and cassette decks, and the uses of each depend on the medium on which they are held. For example, although the access times of registers and RAM chips are exceedingly fast — making them good storage devices for direct access by the processor — they require a permanent charge to hold the information within the cell, otherwise the device will reset to its default state and the information will be lost. On the other hand, disk and tape drives use magnetic media to store information. But although it takes much longer to access the information compared with the microelectronic devices, data is held permanently even after the host computer has been switched off. For this reason, such storage devices are known as 'backing store'.

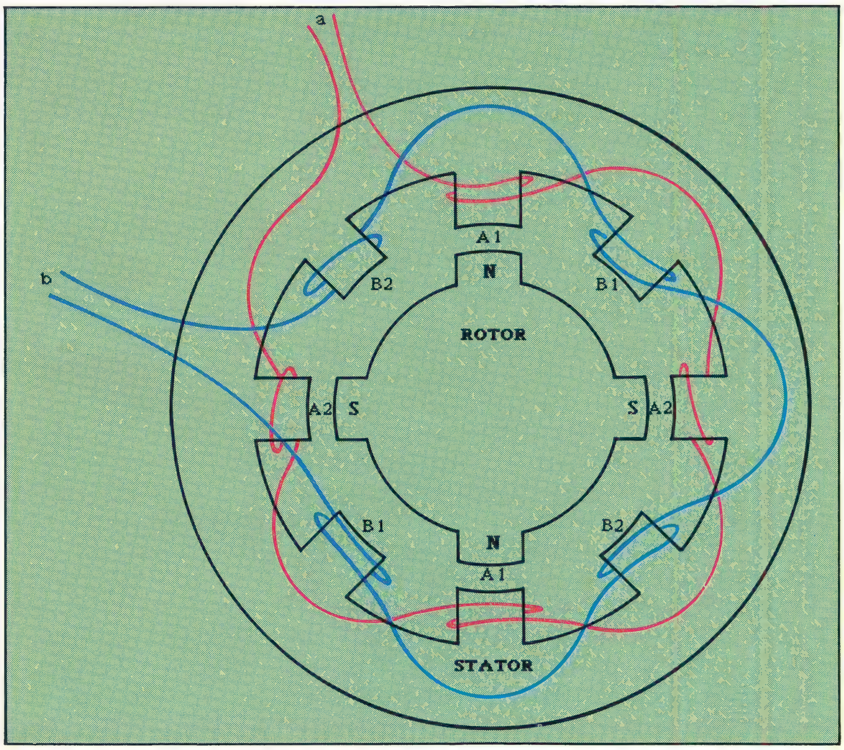
STREAM

Peripheral devices and other I/O systems within a computer often require particular transmission rates and other protocols. To ensure that the information is delivered to a given device in the correct format, it's necessary for the computer's operating system to be organised as a series of *streams*. By assigning information to a particular stream, we tell the computer what format the data should have and where it should be sent.

Although the idea of streams was originally developed to enable computers to handle more than one peripheral interface at once, manufacturers have taken advantage of this simple system of defining output channels to allow a large number of channels to be accessed.

S

Precise Steps
Stepper motors are ideally suited for a wide range of robotics projects, due to the fact that the motor is rotated a precise angle in response to a digital signal from a computer. This means that a computer can accurately position a stepper motor to a very fine degree. Stepper motors are most often used in the construction of robots and other computer-controlled precision instruments such as telescopes



KEVIN JONES

JEWEL OF A SYSTEM

We begin a short series on WIMP systems with a look at 'object-oriented' languages. From their origins in SMALLTALK in the 1970s, we examine the ways in which these operating environments are radically changing the new generation of computers.

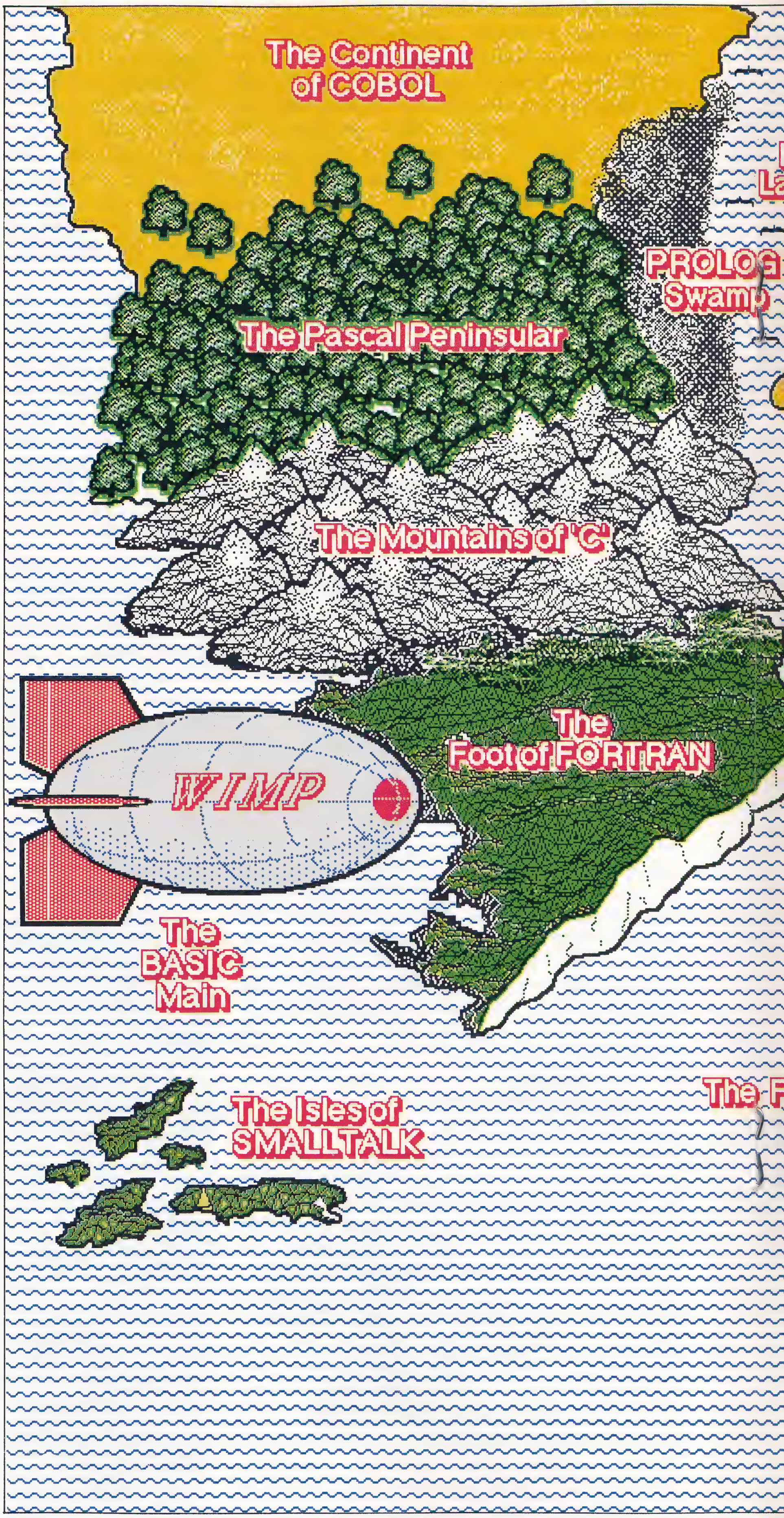
The average newcomer's introduction to personal computers is usually obtained by way of one of two types of program that runs or is 'booted up' when the machine is first switched on. With the first type, machines are provided with a resident version of BASIC — which often gives the impression to naive users that computers are little more than 'BASIC machines' into which commands are entered in direct mode (in other words without program line numbers).

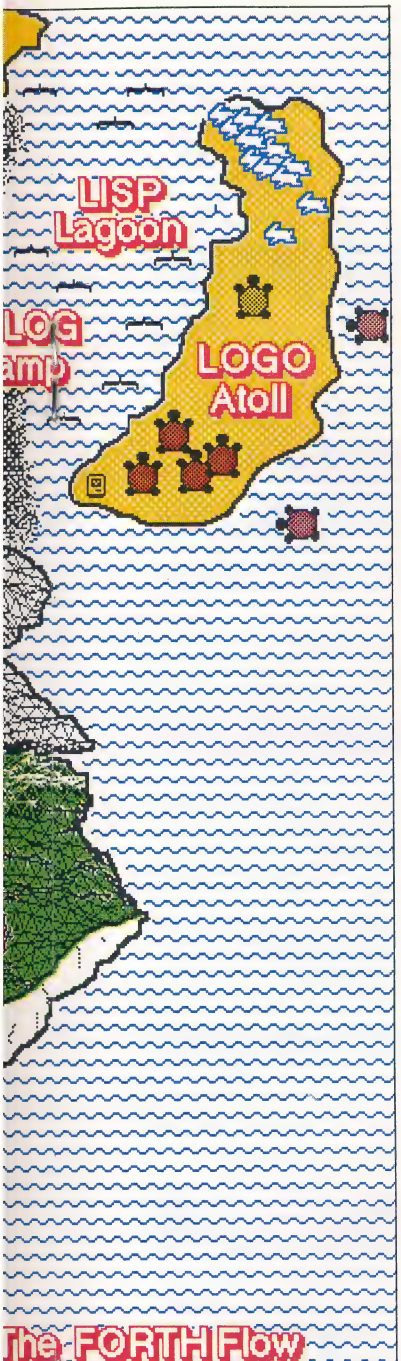
A resident BASIC interpreter, as just described, can provide a more friendly environment than the second type — the 'system program' — which is not a packaged language environment but is known as the monitor, or operating system (OS). Typical examples of standard operating systems are CP/M, MS-DOS, UNIX, and so on. These sytems tend to be rather unfriendly and date back to the days when computer users were expected to be experts in the intricacies of their machine.

As most readers will be aware, machines like the Apple Macintosh have radically altered our expectations of what we get from a computer and especially how we interact with the computer itself. Such systems tend to rely on the use of windows, icons, and mice, and for this reason they have come to be known as WIMP systems. The emergence of WIMP technology is only one aspect of a relatively little-known field of research — usually referred to as 'object-oriented programming' — and the work being done in this area has enormous implications for the future. It is this research, and the changes it has given rise to, that we shall be examining in this series.

The most 'objective' language so far developed is undoubtedly SMALLTALK, the product of a team of researchers at Xerox's Palo Alto Research Centre (PARC). We shall therefore begin our series by taking a look at the background of SMALLTALK and an associated project — Alan Kay's Dynabook.

It may seem strange that a company making most of its profits from paper should evince an interest in the concept of a paperless business environment — the 'electronic office'. This was indeed the case, however, with the Xerox Corporation of America in the early 1970s. Their foresight in anticipating future trends has meant





The 'geography' of programming languages has seen the development of many different environments over the years, each one having its own features and hidden dangers for the unwary traveller. But in a far-flung archipelago known as Smalltalk, only recently discovered by the world at large, there has developed a mode of transport that glides above the hostile environment. The craft, known as The Wimp, sails through a cloudless sky, casting its shadow upon each corner of the land and sea, providing a welcome

IAN McKINNEL ON THE MACINTOSH

that Xerox, perhaps more than any other company, was responsible for the origins of what we now know as WIMP systems.

Xerox realised that with strong competition from IBM, DEC, and others, their late entry into the field of computing would be a serious handicap unless they could capture a share of the market from established computer manufacturers. They took the bold step of setting up PARC, giving its staff of researchers and technicians a broad brief and a large budget to research into office automation, artificial intelligence (AI), the man-made interface (MMI) and computer systems in general.

The starting point for SMALLTALK itself was the futuristic Dynabook, or reference system for everyone. This was the brainchild of Alan Kay, then a research student, who envisaged a time when everyone would have a small self-powered computer (about the size of a novel) that would allow each user dynamic access to a complete reference encyclopaedia of knowledge.

DYNABOOK

The high-performance, hand-held Dynabook would have a high-resolution display, input and output devices supporting both visual and audio communications, and radio satellite network connections to a shared database. Unfortunately, the technical problems of miniaturisation and the massive processing and memory resources required to implement Kay's vision, resulted in the Dynabook research team being disbanded. Nevertheless, many of the Dynabook ideas have had a strong and lasting influence on research and development that has now found its way into currently available machines.

One of the key concepts that evolved during the early development stages was that of the personal 'workstation'. Until then, computing had meant sharing the resources of a large mainframe installation or, more recently, a number of mini-computers. Processing data was very indirect, with 'batch' processing being the norm on large systems.

The most obvious difference between this environment and PARC philosophy was the turn-round time. Conventional systems entailed waiting, sometimes for hours, for results. In 1972 the immediacy that was implicit in the Dynabook concept was quite revolutionary. Furthermore, such concepts as the use of a mouse-like pointing device, icons and windows, meant a radical rethinking about the way data was communicated, both to and from the central processor and the display. This led variously to powerful dedicated graphics kernels implemented in the firmware, and also to the strange language that PARC workers named SMALLTALK.

The core of SMALLTALK research was concerned with modelling a complete hardware and software system based on communication between 'objects' with certain defined properties.

We are all familiar with several distinct objects that form a part of any computer system: the keyboard, VDU and printer are obvious examples. Each of these hardware devices has fixed properties, and only understands how to do certain jobs. There is no use in asking the keyboard to print a document, for instance. Each object, in its own way, holds a certain amount of data (information about itself) and the knowledge required to perform a limited number of tasks correctly. The VDU screen appears to 'know' how to display data sent to it, and take care of its own housekeeping chores, such as sending the cursor back to the left hand side of the screen if text attempts to march off the end of the line.

In reality, this object is a complex interacting network of hardware and software — part electronics, part operating system or other software. By considering every component of a computer system as an object, and defining its properties and the algorithms needed for it to behave correctly, a software engineer can create a system based on a network of communicating objects. Instead of the traditional structured language technique of calling a procedure by name to process data (passed as its parameters), program 'objects' are described according to certain classes or categories, and contain the methods for processing specific data types. These are supplied as arguments in a 'message'. All the message does is to pass on the data and tell the receiving object which method to apply — it is of no concern to the sender of the message exactly how the method is performed..

SMALLTALK is the fullest implementation of the object-oriented approach, but requires large resources of memory and processing power. More recently, there has been considerable interest in MODULA-2, especially in the USA. This European language is a PASCAL derivative and was designed by the inventor of PASCAL, Professor Niklaus Wirth, for programming large systems where a team of programmers must be able to implement different 'modules' independently (hence its name). Data, data types and procedures may be kept separate in any module unless explicitly 'imported' or 'exported' for use by other modules. This is entirely analogous to the SMALLTALK idea, except that the concept of message-passing and data argument is replaced by the more conventional PASCAL-like procedure calls and parameter lists. This provides a simple and secure way to build large systems without the dangerous side-effects that unrestricted access to all objects in the system can produce.

Despite the recent appearance of machines such as the Apple Macintosh, consumers are already taking the immediacy and friendliness of WIMP operating systems for granted. SMALLTALK was responsible for the birth of such systems, and its implementation had important implications for hardware development. In the next instalment, we look at the demands made by WIMP systems on hardware, and the manner in which these were overcome.

FORMULA FOR SUCCESS

We continue our discussion of probability theory with an examination of Bayes' Rule. By utilising these equations to compute the 'likelihood ratios', it is possible to forecast football results.

An Age Of Dilettantism
In the sixteenth and seventeenth centuries, leisure and learning tended to be mixed in a way that might be considered flippant in our own age of specialisation. It's hardly surprising then to find that gambling and mathematics became so frequently entwined. The Reverend Bayes, shown here, developed an equation for the calculation of odds, and other near contemporaries (including D'Alembert, Pascal and Fermat) were equally attracted by the potentially profitable puzzles of probability

Having looked at odds to probability conversion in the previous instalment (see page 1649), we turn now to the application of probability figures, which is where a micro can help the punter.

You can use your computer to work out the probabilities represented by quoted odds, and the bookies can use this to make a nicely over-round book. But most micros are used for trying to pick winners, and the only rational way to go about this is to weigh up the form. The trouble is there's too much information, which gives rise to two problems:

- Sorting out what evidence is important.
- Combining disparate pieces of evidence.

This is where a theorem from statistics, comes in handy — Bayes' Rule. The Reverend Bayes was an eighteenth-century cleric who came up with the following equations:

$$P(H|E)=P(E|H)\times P(H)/P(E)$$

which can be expressed in terms of odds as:

$$O(H|E)=O(H)\times LR(H|E)$$

These may look complicated, but are actually quite straightforward. In the first equation, $P(H|E)$ is the conditional probability, P, of a hypothesis, H, given (|), some evidence, E.

$O(H|E)$ in the second equation is the same as $P(H|E)$, but expressed as odds (in favour). The second equation can be read as: the odds on a hypothesis (H) given a piece of evidence (E) equals the prior odds (before knowing E) on that hypothesis, multiplied by the likelihood ratio for that piece of evidence. We'll discuss likelihood ratios in a moment, but first let's simplify matters by examining a step-by-step example.

FOOTBALL FORECAST

Starting with some sample data, supposing you have records of past football games and you want to predict home wins. Two points are noted: first, when the forecast in *Racing & Football Outlook* newspaper predicts an away win, the game is very seldom a home win (although it may be a draw). Secondly, if the position of the home team (before the match) is in the top nine in its league, the game is usually a home win. With a sample of 131 matches, these observations can be summarised as two frequency tables:

Evidence	Results	
	Home Wins	Away Wins
Paper forecasts away win	3	23
Paper doesn't forecast away win	60	45
Totals	63	68

Evidence	Results		
	Home Wins	Away Wins	Totals
Home team in top nine	31	25	56
Home team lower than ninth	32	43	75
Totals	63	68	131

We can now use these frequency tables to compute the 'likelihood ratios' we mentioned earlier. The likelihood ratio is usually defined as:

$$LR(H|E)=P(E|H)/P(E|\text{not-}H)$$

For two-by-two contingency tables like those above we can compute the likelihood ratios from a table laid out like this:

Evidence	Results		
	Hypothesis	not-Hypothesis	Totals
	a	b	(a+b)
	c	d	(c+d)
Totals	(a+c)	(b+d)	(a+b+c+d)

P(E|H) = number of results that back up evidence/total results in favour of hypothesis; that is, P(E|H=a/(a+c). Similarly, P(E|H=b/(b+d). Our likelihood ratio LR(H|E=a/(a+c)/b/(b+d), with a little juggling comes out as:

$$LR(H|E)=(a\times(b+d)/b\times(a+c))$$

For our two tables, the results come out as follows:

Evidence	LR(Home Win|Evidence)
Paper forecasts away win	0.1408
Paper doesn't forecast away win	1.4392

Evidence	LR(Home Wins|Evidence)
Home team in top nine	1.3384
Home team lower than ninth	0.8032

For those who prefer words to numbers, the latter pair of numbers is saying that if you know that the home team is in the top nine positions of its league, the odds in favour of it winning are 1.3384 times what they would be if you didn't know that fact. But if it is 10th or lower in the league, the odds are only 0.8032 of what they would have been in favour of a home win.

To show these ratios in use, let's assume the first piece of evidence is false (Forecast < > Away) and the second is true (Homepos. < 10). We start with the prior odds in favour of a home which, since there are 63 homes and 68 non-homes (draws or aways), is 63/68 = 0.9265. This corresponds to a probability of 0.4809.

Next, we multiply by the appropriate likelihood ratios. The first item of evidence was false, so we use 1.4392, and the second was true, so we use 1.3384. Thus:

$$\begin{aligned}\text{Posterior Odds} &= 0.9265 \times 1.4392 \times 1.3384\\ &= 1.7846\end{aligned}$$

We can convert back to probability with our formula P=F/(1+F), so:

$$\begin{aligned}\text{Posterior Prob.} &= 1.7846/2.7846\\ &= 0.6409\end{aligned}$$

This says that there is roughly a 64 per cent chance of a home win, given that the newspaper forecast is not for an away and the home team is ninth or better in the league. Both pieces of evidence were favourable, and have raised the prior probability from about 48 per cent to about 64 per cent. If a bookie offers 6 to 4 on, or better, the bet is worth taking (but because of the betting tax, you can't really afford to accept less than evens here).

The advantage of Bayes' Rule is that it provides a rational basis for weighing and combining different items of evidence. It's also easy to compute: once you have the likelihood ratios it's only a matter of multiplication. Furthermore, since the output is in odds, which can readily be converted to probabilities, the rule makes it simple to see whether a bet is good value for money (unlike other systems that come up with 'magic numbers' or strength weightings).

The catch is that the method shown here is likely to produce exaggerated estimates if the pieces of evidence are correlated. If the newspaper pundit, for example, takes into account league position in making his forecast (as is most likely) then the two pieces of evidence are not truly independent. By multiplying them, we are treating them as if they were.

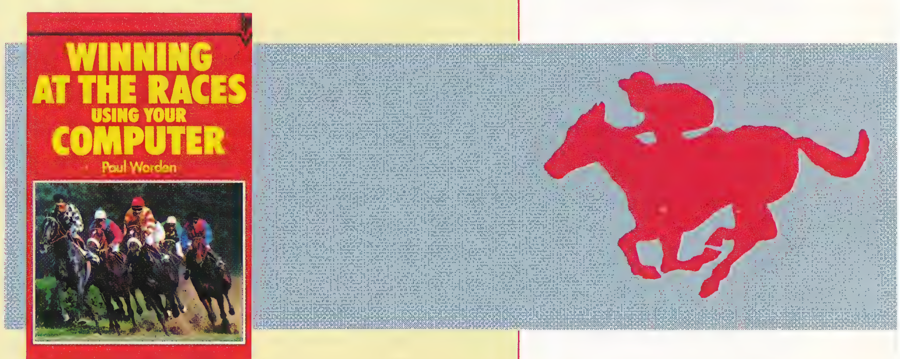
Nevertheless, Bayes' Rule has much to recommend it as a thread for tying together various indicators in a forecasting program. If you write a Bayesian forecasting program, the best way to minimise the problem of correlated evidence is to test new data one at a time, and add a new one only if it improves the overall performance of the system. If a plausible variable, when included, makes the system perform less well, you are probably, in effect, measuring the same thing twice.

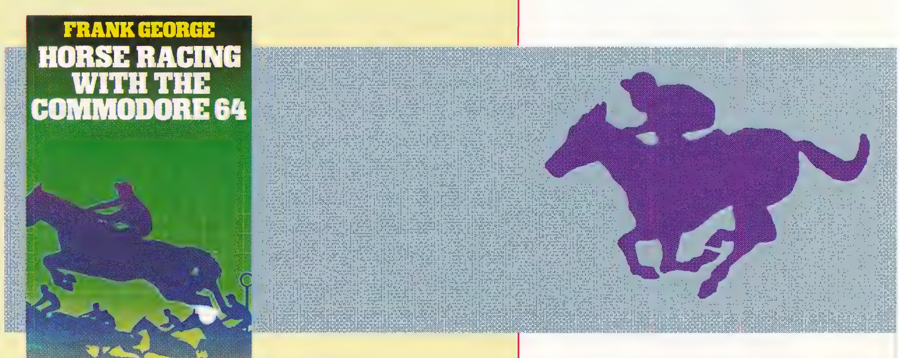
Testing your methods on a sample of past data gets to the heart of the scientific approach to gambling. Unfortunately, this implies some preliminary spadework, which most people tend to skimp on, preferring instead to rely on blind faith.

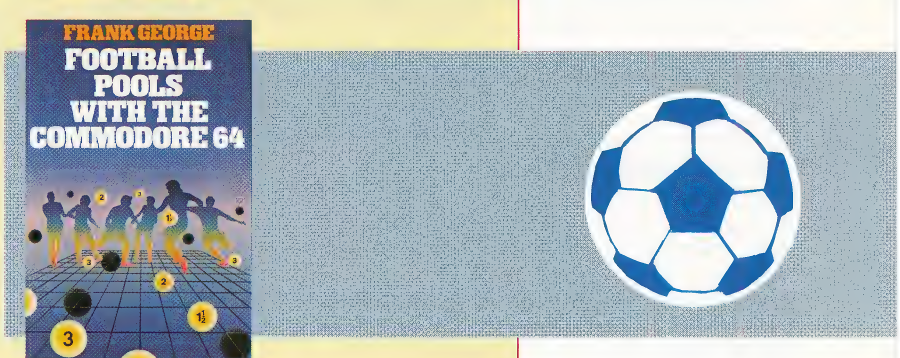
A little common sense, allied to certain basic statistical principles, can help you swing the odds in your favour. The computer can help in this process; but one thing it cannot give you is self-restraint. This is vital if you are to stick to your plans, to approach the matter scientifically and not to bet until you are good and ready.

Further Reading

A number of books on the subject of gambling and computers have already been published. Of the three shown here, Winning at the Races by Paul Worden is available from Interface Publications, 9-11 Kensington High Street, London W8 (£6.95). The two books by Professor Frank George are published by Collins (£7.95) and should now be available at any major bookstore







KEVIN JONES

Basic Flavours

Some flavours have already been printed on pages 1508 and 1605 and readers should enter these as appropriate. In addition, the following flavours should be inserted:

Spectrum:

```
4350 v=INT(RND*a):
    RETURN
4540 RESTORE 9920: FOR
    e=1 TO t(t,n,4): READ h:
    NEXT e: GOSUB h
4570 RESTORE 9930: FOR
    e=1 TO t(t,n,3): READ h:
    NEXT e: GOSUB h:
    RETURN
5470 RESTORE 9910: FOR
    e=1 TO (t(t,n,1)+1):
    READ h: NEXT e: GOTO h
9910 DATA 5480,5490,
    5500,5520,5530,5540,
    5550
9920 DATA 3930,3940,3995
9930 DATA 3030,3060,
    3070,3080,3100,3120,
    3140,3150,3160,3170,
    3180,3190,3200,3210,
    3220,3230,3240,3250,
    3270
```

BBC Micro:

```
4350 v=RND(a): RETURN
```

Commodore 64:

Delete lines 7000 onwards and make the following changes:

```
190 DIM t(5,40,4), k(3,30),
c(35), s(6),
h(6),i$(23)
310 FOR n=1 TO 23: READ
i$(n): NEXT n
4680 m$=i$(t(t,n,4)):
RETURN
```

We present here the entire listing of our interactive characters program, which will run without modification on the Amstrad range of computers. Flavours for the BBC Micro, Sinclair Spectrum and Commodore 64 machines are also included.

This full listing incorporates the remaining decision trees covering object awareness, general activity and character interaction. The listing is printed in two colours — lines printed in black have already been published, lines printed in green are new additions.

There are one or two examples of previously published lines being altered to make room for our new trees — these lines are printed in red.

When RUNning the program, you can enter the character editor at any time by pressing the zero key. Otherwise, pressing one, two or three will move the player into the lounge, saloon and kitchen, respectively.

Dog And Bucket Listing

Initialisation

These lines set up the variables and read data into the different tree arrays

```
10 REM *welcome to the Dog and Bucket*
20 REM
30 REM ..........initialise............
40 REM
45 GOSUB 4030: REM clear the screen
50 DIM l$(3,5),b$(12,4),c$(7,11),d$(11)
60 r=1
70  PRINT "Default values (y/n)?": GOSUB
4110: IF i$="y" OR i$="Y" GOTO 90
80  GOSUB 2350: GOTO 100
90  FOR n=1 TO 7: READ c$(n,1): FOR d=2
TO 11: READ c$(n,d): NEXT d: NEXT n
100 FOR n=1 TO 3: READ l$(n,1): FOR e=2
TO 5: READ l$(n,e): NEXT e: NEXT n
110 FOR n=1 TO 12: READ b$(n,1): FOR d=2
 TO 4: READ b$(n,d): NEXT d: NEXT n
120 FOR n=2 TO 11: READ d$(n): NEXT n
130 DEF FNb(y,z)=VAL(b$(y,z))
140 DEF FNc(y,z)=VAL(c$(y,z))
150 DEF FNm$(c$,d)=STR$((VAL(c$))-d)
160 DEF FNi$=b$(VAL(c$(c,3)),1)
180 REM setup trees
190 DIM t(5,40,4),k(3,30),c(35),s(6),h(6
): z=0
200 REM object tree
210 FOR n=1 TO 21: REM 21 choice nodes
220 READ k(1,n),t(1,n,2),t(1,n,1): NEXT
n
230 REM plot tree
240 FOR n=1 TO 22: FOR s=1 TO 4: READ t(
2,n,s): NEXT s: READ a$: NEXT n
250 REM character interaction tree
260 FOR n=1 TO 22: FOR s=1 TO 4: READ t(
3,n,s): NEXT s: READ a$: NEXT n
270 REM general activity tree
280 FOR n=1 TO 39: FOR s=1 TO 4: READ t(
4,n,s): NEXT s: READ a$: NEXT n
290 REM object awareness tree
300 FOR n=1 TO 13: FOR s=1 TO 4: READ t(
5,n,s): NEXT s: READ a$: NEXT n
500 REM
510 REM test program loop
520 REM
530 GOSUB 2100: GOSUB 2150: GOSUB 2240:
PRINT: PRINT: GOSUB 1000: GOTO 530
1000 REM
1010 REM character handler
1020 REM
1030 REM check to see if key pressed
1040 GOSUB 4260: IF i$<>"" THEN GOSUB 20
40: RETURN
1050      REM process each character in t
urn
1060 FOR c=1 TO 6: IF c$(c,9)="7" THEN c
$(c,9)="0"
1070 IF z=c THEN GOTO 1500
1080 IF FNc(c,10)>0 THEN c$(c,10)=FNm$(c
$(c,10),1): GOTO 1500
1090      REM flag=0 so reset flag and pr
ocess character
1100 RESTORE: FOR n=1 TO c*10+c-1: READ
c$(c,10): NEXT n
1110 IF FNc(c,10)=0 THEN GOTO 1500: REM
default value=0 so don't process
1120      REM check move flag
1130 IF FNc(c,11)>0 THEN c$(c,11)=FNm$(c
$(c,11),1): GOTO 1190
```

Character Handler

Lines 1000 to 1500 check each character in turn and decide whether or not it should be processed or moved from one location to another

```
1140      REM move flag=0 so reset flag a
nd move character
1150 RESTORE: FOR n=1 TO c*11: READ c$(c
,11): NEXT n
1160 IF c$(c,11)="0" THEN GOTO 1180
1170 GOSUB 2840: GOTO 1500
1180      REM sort through the trees
1190 GOSUB 2400: REM initialise conditio
ns
1200 FOR t=2 TO 5: GOSUB 2430: GOSUB 546
0: NEXT t
1210 IF FNc(c,4)>0 THEN GOSUB 5000: REM
object manipulation tree
1500 NEXT c: GOTO 1030: REM do next char
acter - when all finished check for key
press/do again
```

High-Level Routines
These routines check for characters and objects, print their details to the screen, and perform other general-purpose functions

```
2000 REM
2010 REM input routine
2020 REM
2030 GOSUB 4110
2040 IF (ASC(i$)<48) OR (ASC(i$)>51) THE
N RETURN
2050 IF i$="0" THEN GOSUB 2320: RETURN
2060 r=VAL(i$): c$(7,2)=i$: RETURN
2070 REM
2080 REM location print
2090 REM
2100 PRINT l$(r,1)
2110 RETURN
2120 REM
2130 REM print visible objects
2140 REM
2150 PRINT "You can see: ";
2160 p=0: FOR b=1 TO 12: IF VAL(b$(b,2))
<>r GOTO 2190
2170 p=p+1: IF p>1 THEN PRINT ", ";
2180 PRINT b$(b,1);
2190 NEXT b
2200 IF p=0 THEN PRINT "nothing";
2210 PRINT: RETURN
2220 REM
2230 REM print visible characters
2240 REM
2250 p=0: FOR c=1 TO 6: IF VAL(c$(c,2))<
>r GOTO 2290
2260 p=p+1: IF p=1 THEN PRINT "You are i
n the company of: ";: GOTO 2280
2270 PRINT ", ";
2280 PRINT c$(c,1);
2290 NEXT c
2300 IF p=0 THEN PRINT "There's no-one h
ere."
2310 RETURN
2320 REM
2330 REM initialise chars subroutine
2340 REM
2350 PRINT: h=c: RESTORE: FOR c=1 TO 7:
READ c$(c,1): GOSUB 4070: PRINT c$(c,1);
" - ";: PRINT "Set this char?": GOSUB 41
10
2360 IF (i$<>"y") AND (i$<>"Y") THEN FOR
 n=2 TO 11: READ n$: NEXT n: GOTO 2390
2370 FOR d=2 TO 11: READ s$: PRINT d$(d)
;: INPUT i$: IF i$<>"" THEN c$(c,d)=i$
2380 NEXT d
2390 NEXT c: c=h: RETURN
2400 REM
2410 REM conditions
2420 REM
2430 h=FNc(c,8): i=FNc(c,3): j=FNc(c,6):
 c(1)=ABS(i>0): c(2)=ABS((FNb(j,2)=FNc(c
,2)) AND (q=1)): c(3)=ABS(b$(i,3)="y"):
c(4)=ABS(i=j): c(5)=ABS(b$(i,4)="y")
2440 c(6)=ABS(i=3): c(7)=ABS(FNc(c,5)>5)
: c(8)=ABS(FNc(c,5)>2): c(9)=ABS(VAL(c$(
c,9))=1): c(10)=ABS(FNc(x,3)=0): c(11)=A
BS(FNc(h,2)=FNc(c,2)): c(12)=255
2450 c(13)=ABS(z=c): c(14)=ABS(g=c): c(1
5)=ABS(z=0): c(16)=ABS(s(c)=255): c(17)=
ABS(FNc(z,2)=FNc(c,2)): c(18)=ABS(h(c)=2
55): c(19)=ABS(i=2)
2460 c(20)=ABS(FNc(x,4)<0): c(21)=ABS(z=
x): c(22)=ABS(FNc(x,3)=FNc(c,6)): c(23)=
ABS(FNc(x,5)>15): c(24)=ABS(FNc(x,9)=7):
 c(25)=ABS(FNc(c,5)>15): c(26)=ABS(FNc(c
,5)>17)
2470 c(27)=FNc(c,9): c(28)=ABS(FNc(c,2)=
r): c(29)=ABS(FNc(c,2)=1): c(30)=ABS(FNc
(c,5)<5)
2500 RETURN
2510 REM
2520 REM check location for objects
2530 REM
2540 f=0: REM set 'found flag' to zero
2550 FOR b=1 TO 12
2560 IF FNb(b,2)=FNc(c,2) THEN f=1: b=12
2570 NEXT b
2580 IF f=1 THEN n=3: GOTO 2600
2590 n=39
2600 RETURN
2610 REM
2620 REM check for presence of owner of
object: if present, set x to character n
umber: jump back into tree
2630 REM
2640 f=0:x=0
2650 FOR m=1 TO 6
2660 IF (FNc(m,2)=FNc(c,2)) AND (FNc(m,6
)=FNc(c,3)) THEN f=1: x=m: GOSUB 2430: m
=6
2670 NEXT m
2680 IF f=1 THEN n=15: GOTO 2700
2690 n=39
2700 RETURN
2710 REM
2720 REM select object at random from ch
aracter's location
2730 REM
2740 b=0
2750 FOR s=1 TO 12
2760 IF FNb(s,2)<>FNc(c,2) THEN GOTO 278
0
2770 GOSUB 4180: IF q=1 THEN b=s: s=12
2780 NEXT s
2790 IF b=0 THEN GOTO 2750
2800 RETURN
2810 REM
2820 REM move a character
2830 REM
2840 IF FNc(c,4)<1 THEN RETURN: REM too
weak to move
2850 y=0: f=0: FOR w=2 TO 5: IF l$(VAL(c
$(c,2)),w)="0" THEN GOTO 2910
2860 GOSUB 4180: IF q=1 THEN f=1: c$(c,9
)="7": GOTO 2880
2870 GOTO 2910
2880 IF FNc(c,2)=r THEN PRINT c$(c,1);"
leaves the room...";: y=1
2890 c$(c,2)=l$(VAL(c$(c,2)),w): w=5: IF
 FNc(c,2)=r THEN PRINT c$(c,1);" enters
the room...";: y=1
2900 IF y=1 THEN y=c: GOSUB 2250: c=y: P
RINT: PRINT: REM update characters prese
nt message
2910 NEXT w: IF f=0 GOTO 2850
2920 RETURN
```

Action Tables
Lines 3000 to 3999 hold the various routines called during execution of the different decision trees

```
3000 REM
3010 REM action table
3020 REM
3030 FOR n=1 TO 3: GOSUB 4090: NEXT n: m
$=c$(c,1)+" takes another look at the bo
dy, and suddenly realises the hideous tr
uth. "+CHR$(34)+"The pasty's made of cat
-food"+CHR$(34)+" ": GOSUB 4630
3040 GOSUB 4390: m$=m$+"yells, and in a
mad rush the assembled company scramble
over the bar and attack Fred the Barman,
 who pleads with them to no avail to spa
re his miserable life.": GOSUB 4630: GOS
UB 4720
3050 FOR n=1 TO 2000: NEXT n: GOSUB 4720
: m$="...and the next day Fred the Barma
n is nowhere to be seen.  However, a num
ber  of oddly shaped pasties are availab
le to feed the ever-hungry clientele of
the Dog and Bucket...": GOSUB 4630: GOSU
B 4720: END
3060 h(c)=255: GOSUB 4680: m$=c$(c,1)+m$
: GOSUB 4630: GOSUB 4720: RETURN
3070 z=c: GOSUB 4680: n$=c$(c,1)+m$: GOS
UB 4300: m$=n$+m$+"stomach, and immediat
ely expires.  The other characters are t
oo involved with their drinks to notice.
..": GOSUB 4630: GOSUB 4720: g=0: c$(c,4
)="-1": RETURN
3080 c$(c,4)="10": g=0: IF t(t,n,4)>0 TH
EN GOSUB 4680: m$=c$(c,1)+m$: GOSUB 4630
: GOSUB 4720
3090 RETURN
3100 a=20: GOSUB 4350: IF v<>5 THEN RETU
RN
3110 GOSUB 4680: GOSUB 4630: GOSUB 4720:
 RETURN
3120 a=2: GOSUB 4350: IF v<>0 THEN RETURN
```

```
3130 m$=c$(c,1)+" tries to wake "+c$(x,1
)+" without success...": GOSUB 4630: GOS
UB 4720: RETURN
3140 GOSUB 4680: m$=CHR$(34)+m$+CHR$(34)
+" says "+c$(c,1)+" to "+c$(x,1): GOSUB
4630: GOSUB 4720: RETURN
3150 GOSUB 4680: m$=CHR$(34)+m$+CHR$(34)
+", "+c$(c,1)+" asks "+c$(x,1): GOSUB 46
30: GOSUB 4720: RETURN
3160 GOSUB 4680: m$=c$(c,1)+" and "+c$(x
,1)+m$: GOSUB 4630: GOSUB 4720: c$(c,5)=
"10": RETURN
3170 GOSUB 4680: m$=CHR$(34)+m$+CHR$(34)
+" says "+c$(c,1)+" to "+c$(x,1): GOSUB
4630: GOSUB 4720: RETURN
3180 GOSUB 4680: m$=c$(c,1)+m$: GOSUB 46
30: GOSUB 4720: c$(c,5)="5": RETURN
3190 GOSUB 4680: m$=c$(c,1)+m$: GOSUB 46
30: GOSUB 4720: RETURN
3200 GOSUB 4680: m$=CHR$(34)+m$+CHR$(34)
+" says "+c$(c,1)+" dejectedly...": GOSU
B 4630: GOSUB 4720: c$(c,5)="10": RETURN
3210 GOSUB 4680: m$=CHR$(34)+m$+CHR$(34)
+" says "+c$(c,1): GOSUB 4630: GOSUB 472
0: c$(c,9)="3": RETURN
3220 x=FNc(c,8): m$=c$(c,1)+" and "+c$(x
,1)+" are deep in conversation.": GOSUB
4630: GOSUB 4720: GOSUB 4220: x=0: RETUR
N
3230 x=FNc(c,3): IF x>0 THEN GOSUB 4680:
 m$=c$(c,1)+" drunkenly examines "+b$(x,
1)+m$: GOSUB 4630: GOSUB 4720: GOSUB 422
0: x=0
3235 RETURN
3240 GOSUB 4680: m$=CHR$(34)+m$+CHR$(34)
+" yelps "+c$(c,1): c$(c,5)="4": GOSUB 4
630: GOSUB 4720: GOSUB 4220: RETURN
3250 GOSUB 4680: m$=CHR$(34)+m$+CHR$(34)
+" howls "+c$(c,1)+", looking at the ":
GOSUB 4630: IF FNc(c,3)=3 THEN m$="pasty
.": GOSUB 4630: GOSUB 4720: GOSUB 4220:
RETURN
3260 m$="sandwich.": GOSUB 4630: GOSUB 4
720: GOSUB 4220: RETURN
3270 GOSUB 4680: m$=c$(c,1)+m$: GOSUB 46
30: GOSUB 4300: GOSUB 4630: m$="drink":
GOSUB 4630: GOSUB 4720: RETURN
3900 REM
3910 REM gosub table
3920 REM
3930 s(c)=255: m$=c$(c,1)+" kneels down
beside the prostrate body of "+c$(z,1)+"
. The horrible truth slowly dawns, but
the others seem too drunk to pay any imm
ediate attention...": GOSUB 4630: GOSUB
4720: RETURN
3940 f=0:x=0: FOR y=1 TO 6
3950 IF y=c THEN GOTO 3970
3960 IF FNc(y,2)=FNc(c,2) THEN f=1: GOSU
B 4180: IF q=1 THEN x=y: GOSUB 2430: y=6
3970 NEXT y: IF f=0 THEN t(3,1,4)=2: RET
URN
3980 IF x=0 THEN GOTO 3940
3990 RETURN
3995 c$(c,5)=FNm$(c$(c,5),1): RETURN
```

Low-Level Routines
These routines output data to the screen, and take care of other low-level functions, such as sounding a BEEP and getting characters from the keyboard

```
4000 REM
4010 REM low level system subroutines
4020 REM
4030 REM clear the screen
4040 REM
4050 CLS: RETURN
4060 REM
4070 REM beep
4080 REM
4090 PRINT CHR$(7);: RETURN
4100 REM
4110 REM get a character from the keyboa
rd
4120 REM
4130 i$=INKEY$: IF i$="" GOTO 4130
4140 RETURN
4150 REM
4160 REM random number routine
4170 REM
4180 q=INT(RND(1)*2)+1: RETURN
4190 REM
4200 REM zero character codes
4210 REM
4220 c$(c,8)="0": c$(c,9)="0": RETURN
4230 REM
4240 REM test to see if key pressed
4250 REM
4260 i$=INKEY$: RETURN
4270 REM
4280 REM print his/her
4290 REM
4300 IF c$(c,7)="f" THEN m$="her ": RETU
RN
4310 m$="his ": RETURN
4320 REM
4330 REM variable random number routine
4340 REM
4350 v=INT(RND(2)*a): RETURN
4360 REM
4370 REM print he/she
4380 REM
4390 IF c$(c,7)="f" THEN m$="she ": RETU
RN
4400 m$="he ": RETURN
4500 REM
4510 REM jumpblock
4520 REM
4530 REM re-entrant nodes
4540 ON t(t,n,4) GOSUB 3930,3940,3995
4550 RETURN
4560 REM action nodes
4570 ON t(t,n,3) GOSUB 3030,3060,3070,30
80,3100,3120,3140,3150,3160,3170,3180,31
90,3200,3210,3220,3230,3240,3250,3270: R
ETURN
4600 REM
4610 REM print messages if player presen
t
4620 REM
4630 IF FNc(c,2)=r THEN PRINT m$;
4640 m$="": RETURN
4650 REM
4660 REM select a message from data stat
ement
4670 REM
4680 RESTORE 7030: FOR m=1 TO t(t,n,4):
READ m$: NEXT m: RETURN
4690 REM
4700 REM print a blank line
4710 REM
4720 IF FNc(c,2)=r THEN PRINT: PRINT
4730 RETURN
```

Tree Traversal
Lines 5000 to 5999 sort the different trees. Routines for the object manipultion tree are also included here

```
5000 REM object tree routines
5010 p=0: REM zero print flag
5020 IF FNc(c,2)=r THEN p=1
5030 n=1: REM start at node 1
5040 IF n>21 GOTO 5070
5050 k=c(k(1,n))+1: IF k(1,n)=12 THEN GO
SUB 4180: k=q
5060 n=t(1,n,k): GOTO 5040
5070 IF n>=24 GOTO 5090
5080 ON (n-21) GOSUB 2540,2640: GOTO 504
0
5090 ON (n-23) GOTO 5100,5130,5160,5180,
5210,5240,5260,5270,5280,5300,5310,5330,
5340,5360,5370,5430
5100 GOSUB 2740: c$(c,3)=STR$(b)
5110 IF p=1 THEN PRINT c$(c,1);" picks u
p";b$(b,1): PRINT
5120 b$(b,2)="0": c$(c,9)="4":  RETURN
5130 c$(c,3)=c$(c,6)
5140 IF p=1 THEN PRINT c$(c,1);" picks u
p";FNi$: PRINT
5150 b$(VAL(c$(c,3)),2)="0": c$(c,9)="4"
:  RETURN
5160 IF p=1 THEN PRINT c$(c,1);" takes a
 sip from";FNi$: PRINT
5170 c$(c,4)=FNm$(c$(c,4),-1): RETURN
5180 GOSUB 4180: IF (p=1) AND (q=1) THEN
 PRINT c$(c,1);" is eating the sandwich.
": PRINT
5190 c$(c,4)=FNm$(c$(c,4),-2): c$(c,9)="
6": GOSUB 4180: IF q=1 THEN GOSUB 4220
5200 RETURN
5210 IF p=1 THEN PRINT c$(c,1);" takes a
 tentative bite of the pasty, groans, an
d drops it on the floor.": PRINT
5220 g=c: REM set pasty eaten flag
5230 c$(c,3)="0": c$(c,4)=FNm$(c$(c,4),1
0): b$(3,2)=c$(c,2): RETURN
5240 IF p=1 THEN PRINT c$(c,1);" puts do
wn";FNi$: PRINT
5250 b$(VAL(c$(c,3)),2)=c$(c,2): c$(c,3)
="0": RETURN
5260 c$(c,5)=FNm$(c$(c,5),-1): RETURN
5270 GOSUB 5240: RETURN
5280 IF p=1 THEN PRINT c$(c,1);" throws"
;b$(VAL(c$(c,3)),1);" at ";c$(x,1): PRIN
T
```

```
5290 c$(x,4)=FNm$(c$(x,4),1): b$(VAL(c$(
c,3)),2)=c$(c,2): c$(x,8)=STR$(c): c$(x,
9)="5": c$(c,3)="0": RETURN
5300 GOSUB 4220: RETURN
5310 IF p=1 THEN PRINT "I think I've got
 your drink, says ";c$(c,1);" to ";c$(x,
1): PRINT
5320 c$(c,8)=STR$(x): c$(c,9)="2": RETUR
N
5330 c$(c,4)=FNm$(c$(c,4),2): RETURN
5340 IF p=1 THEN PRINT c$(c,1);" gives";
FNi$;" to ";c$(x,1): PRINT
5350 c$(x,3)=c$(c,3): c$(c,3)="0": c$(c,
8)=STR$(c): c$(c,9)="1": RETURN
5360 GOSUB 4220: RETURN
5370 IF p=0 GOTO 5420
5380 IF p=1 THEN PRINT c$(c,1);" is drun
kenly thanking ";c$(VAL(c$(c,8)),1);" fo
r returning ";
5390 IF p=1 AND c$(c,7)="f" THEN PRINT "
her ";: GOTO 5410
5400 PRINT "his ";
5410 PRINT "drink": PRINT
5420 GOSUB 4220: RETURN
5430 RETURN
5440 REM
5450 REM sort trees
5460 n=1
5470 ON (t(t,n,1)+1) GOTO 5480,5490,5500
,5520,5530,5540,5550
5480 k=c(t(t,n,2))+1: n=t(t,n,2+k): GOTO
 5470
5490 GOSUB 4530 : n=t(t,n,3): GOTO 5470
5500 GOSUB 4570
5510 RETURN
5520 GOSUB 4680: GOSUB 4630: GOSUB 4720
5530 RETURN
5540 a=t(t,n,4): GOSUB 4350: n=t(t,n,3)+
v: GOTO 5470
5550 k=c(t(t,n,2)): n=t(t,n,3)+k: GOTO 5
470
```

Data Store
Lines 6000 onward hold the data for the various arrays, as well as messages used by the different program modules

```
6000 REM
6010 REM character data
6020 REM
6030 DATA "Toby Belcher","2","7","10","1
0","7","m","0","0","1","4","Fiona Frappe
","1","8","30","10","8","f","0","0","1",
"5","Steve Swig","1","9","8","10","9","m
","0","0","1","6","Sally Short","2","0",
"20","10","10","f","0","0","1","5"
6040 DATA "Rupert Beer","2","11","10","6
","11","m","0","0","1","6","Molly Mixer"
,"1","12","15","6","12","f","0","0","1",
"5","you","1","0","255","255","0","m","0
","0","0","0"
6050 REM
6060 REM location data
6070 REM
6080 DATA "You are in the lounge of the
Dog and    Bucket. Several shady charact
ers are    gathered together in a corner
 playing   dominoes.  Behind the counter
, Fred the Barman looks his usual cheery
 self.     Exits lead east.","0","0","2"
,"0"
6090 DATA "Behold the saloon of the Dog
and Bucket,which looks as if it could do
 with      extensive redecoration. The f
loor       appears to have been regularl
y hosed    down with beer slops.  Doors
lead west  and south.","0","3","0","1"
6100 DATA "Ugh! This is the kitchen, whe
re the     famous Dog and Bucket Cornish
 Pasties   are prepared for an ever-hung
ry         clientele. You notice a numbe
r of empty cat-food tins, which is stran
ge because there are no cats.","2","0","
0","0"
6110 REM
6120 REM object data (for b$(12,4))
6130 REM
6140 DATA " a glass of beer","2","n","y"
," an empty tin of catfood","3","n","n",
" a Dog and Bucket cornish pasty","1","y
","n"," a bar-stool","2","n","n"," an as
htray","1","n","n"
6150 DATA " a stale ham sandwich","2","y
","n"," a pint of bitter","0","n","y","
a creme de menthe","0","n","y"," a whisk
y and water","0","n","y"," a neat vodka"
,"2","n","y"," a pint of lager","0","n",
"y"," a qin and ginger ale","0","n","y"
6160 REM
6170 REM character attribute data (for d
$(11))
6180 REM
6190 DATA "Location","Strength","Invento
ry","Mood","Own object","Sex","Last char
 (lch)","Last command code (lcd)","Handl
e frequency","Move frequency"
6200 REM
6210 REM object tree data
6220 REM
6230 DATA 1,2,22,12,5,4,2,7,6,3,9,8,4,11
,10,12,39,24,12,6,25,5,12,39,6,13,27,12,
23,29,12,30,14,12,26,39,12,28,39,12,31,1
7,7,18,16,8,39,19,9,21,39,10,36,20,12,33
,32,12,35,34,11,38,37
6240 REM
6250 REM plot tree data
6260 REM
6270 DATA 0,13,2,22," ",0,14,5,3," ",0,1
5,21,4," ",5,0,18,3," ",0,16,6,7," ",0,1
7,7,11," ",0,18,9,8," ",0,17,12,13," ",0
,19,10,17," ",5,0,14,3," ",1,0,7,1," ",4
,0,0,0," ",2,0,1,0," ",2,0,5,1," ",4,0,0
,0," ",4,0,0,0," ",2,0,2,4," ",2,0,3,2,"
 ",2,0,4,3
6280 DATA " ",2,0,4,0," ",4,0,0,0," ",4,
0,0,0," "
6300 REM interaction tree
6320 DATA 1,0,3,2," ",4,0,0,0," ",0,20,6
,4," ",0,21,7,5," ",4,0,0,0," ",5,0,8,3,
" ",2,0,6,0," ",0,22,11,14," ",0,23,12,1
7," ",0,24,13,18," ",4,0,0,0," ",4,0,0,0
," ",4,0,0,0," ",5,0,15,2," ",4,0,0,0,"
",2,0,7,5," ",0,25,22,21," "
6330 DATA 5,0,19,2," ",4,0,0,0," ",2,0,8
,6," ",2,0,9,7," ",2,0,10,8," "
6340 REM general activity tree
6350 DATA 5,0,2,5," ",0,26,11,7," ",6,27
,24,7," ",0,28,8,12," ",0,29,9,15," ",5,
0,21,3," ",2,0,11,9," ",4,0,0,0," ",4,0,
0,0," ",4,0,0,0," ",0,30,10,18," ",5,0,1
3,2," ",2,0,12,10," ",4,0,0,0," ",5,0,16
,2," ",3,0,0,11," ",3,0,0,12," "
6360 DATA 5,0,19,2," ",4,0,0,0," ",2,0,1
3,13," ",2,0,14,14," ",2,0,14,15," ",2,0
,14,16," ",4,0,0,0," ",0,11,31,32," ",4,
0,0,0," ",5,0,33,3," ",0,11,36,37," ",5,
0,38,2," ",2,0,14,17," "
6370 DATA 4,0,0,0," ",2,0,15,0," ",4,0,0
,0," ",2,0,16,18," ",4,0,0,0," ",4,0,0,0
," ",2,0,17,19," ",4,0,0,0," ",2,0,18,20
," "
6380 REM object awareness tree
6390 DATA 0,4,8,2," ",5,0,3,5," ",4,0,0,
0," ",4,0,0,0," ",4,0,0,0," ",4,0,0,0,"
",2,0,14,21," ",1,0,9,3," ",5,0,10,4," "
,2,0,19,22," ",2,0,14,23," ",4,0,0,0," "
,4,0,0,0," "
7000 REM
7010 REM message data
7020 REM
7030 DATA "A strange smell fills the air
...could it be the odour of Catty-Kit A
La Carte?"," suddenly collapses on the f
loor, clutching "," looks rather ill, an
d warns the others not to touch the past
y."
7040 DATA " examines the tin carefully,
 and assumes a thoughtfull expression.","
What are you doing with my drink?","Wher
e have you been?"
7050 DATA ", in a drunken fit, jump onto
 the table and dance together."
7060 DATA "You look cheerful"," clambers
 drunkenly onto the bar, tries to dance,
 and falls off again."
7070 DATA ", in a drunken stupor, sudden
ly looks up and peers through the VDU sc
reen at you....","Fred the Barman pulls
another pint...","Fred is busy cleaning
glasses"
7080 DATA "It's a dog's life...","I come
 not to bury Caesar...","Let me tell you
 the story of my life...","Barman, fill
my glass!"
7090 DATA "Hi there, everyone..."," as i
f it held the secret of life.","What did
 you do that for!!","I think I'm going t
o be sick!"
7100 DATA "Ahh...This drink is sublime..
."," is desperately hunting for ","Who's
 got my drink?"
```

CHANGE OF KEY

The keyboard of the Amstrad may easily be configured to suit the needs of any application. We look at the ways these changes may be implemented through the operating system's key manager.

The Amstrad operating system has one section, called the key manager, that controls all access to the keyboard and the joystick. The real power of the key manager lies in the fact that the keyboard is entirely 'soft'. This means that the code returned from each physical key is user-definable. This approach makes installation of programs such as CP/M easy, as keys may be tailored to give the required codes rather than modifying the program to accept the codes produced by default.

There are two types of code that may be returned from a keypress — either a single ASCII character or an expansion token. An expansion token can be thought of as a flag to return a string of characters from a single keypress, rather than an individual character.

The keyboard is sampled every one-fiftieth of a second on the general-purpose ticker. A key state map is maintained to keep a record of which keys were pressed on the last sampling. This map is used to ensure the keys are not read more than once and to determine whether or not a character should be repeated. A buffer is also maintained that stores the codes returned by the keys at the time their presence was first detected; these may be interpreted as either ASCII characters or expansion tokens.

THE KEY MANAGER

The key manager may be accessed at several different levels. The lowest level allows the user to test whether a physical key is being pressed at the current time.

The second level returns the single character that has been assigned to a key. This character will either be an ASCII value or an expansion token. If it is an expansion token, then it is up to the user to determine what its interpretation is. This level takes into account the current state of the Shift and Control keys when determining which character should be returned.

The highest level simply returns the next character that has been typed at the keyboard. This character may either correspond to a single keypress or it may be the next character from an expansion token.

Associated with each key are three translation tables that determine the code returned when a keypress is detected in its normal, shifted and control states. Each of these entries may be determined and set individually, using the entries detailed in the diagram.

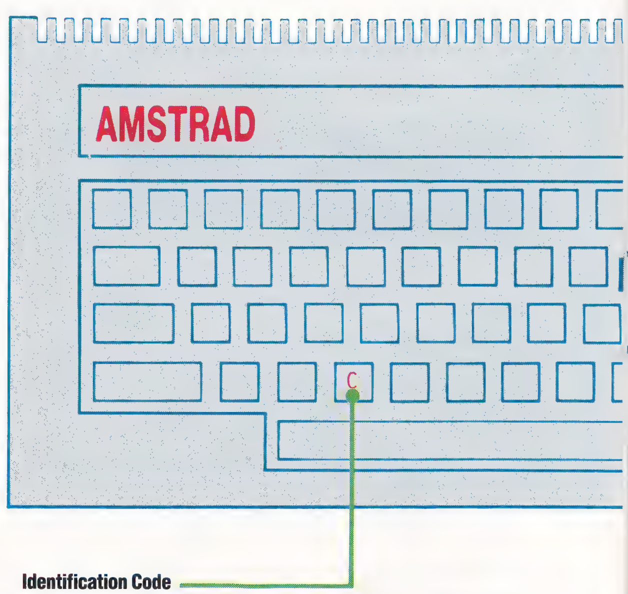
The value that may be assigned to each state can range from 0-255 and is normally interpreted as a character code. However, certain codes have particular meanings as detailed in the diagram. Note that characters from &E0 to &FC have special meaning under BASIC and so if a program assigns a key to return one of these tokens, it should reassign them before returning to BASIC.

The first listing shows how a set of translations can be set up on entry to a routine, while at the same time saving their initial states. This then allows the keys to be reassigned on exit from the routine so that the translations become transparent to the main program.

EXPANSION TOKENS

Codes &80 to &9F are defined as expansion tokens. When they are encountered they are stored in the buffer unless they are read out using

Keyboard Management



Identification Code
Each key is assigned a unique key number that cannot be altered by the firmware. The ASCII code returned by a key is determined by three tables, which return the code for the key in its normal, shifted, and control state

CAROLINE CLAYTON

KM_READ_CHAR or KM_WAIT_CHAR. In this case, the key manager looks to determine which string is assigned to that expansion token and inserts the characters in sequence into the buffer until the end of the string.

The expansion strings are stored in a buffer, which can be of any size depending on the number of strings required. Each string can consist of up to 255 characters providing there is sufficient room in the buffer which must lie in the central 32 Kbytes of RAM. The buffer size and location is set using KM_EXP_BUFFER.

Each expansion token may have a string assigned to it using KM_SET_EXPAND. The characters within the string are not checked by the firmware, so it's possible to return characters such as &FC (the break token) to a routine. The strings assigned to an expansion token can only be examined one character at a time, using the KM_GET_EXPAND entry.

Every key on the keyboard has a number associated with it. This number is unique and cannot be altered via the firmware, thus providing an absolute method of referring to a key. The key numbers are detailed in Appendix III of the firmware manual and are also thoughtfully provided on the disk-drive cases of the CPC 664 and the 6128.

The KM_TEST_KEY entry is the lowest level entry into the key manager. It allows the user to determine whether a particular key was held down on the last keyboard scan, with the exception of the Control and Shift keys. This entry is particularly useful when only one key is being tested; for example, when the keyboard is being polled periodically for the Escape key to signify that the program should be terminated.

An entry is also provided exclusively for the joysticks: KM_GET_JOYSTICK. This entry reads the joystick states from the key state map and is the fastest method of determining their current status. It's therefore particularly suited to games programmers, as joysticks may be read at any time regardless of whether or not their positions have been altered.

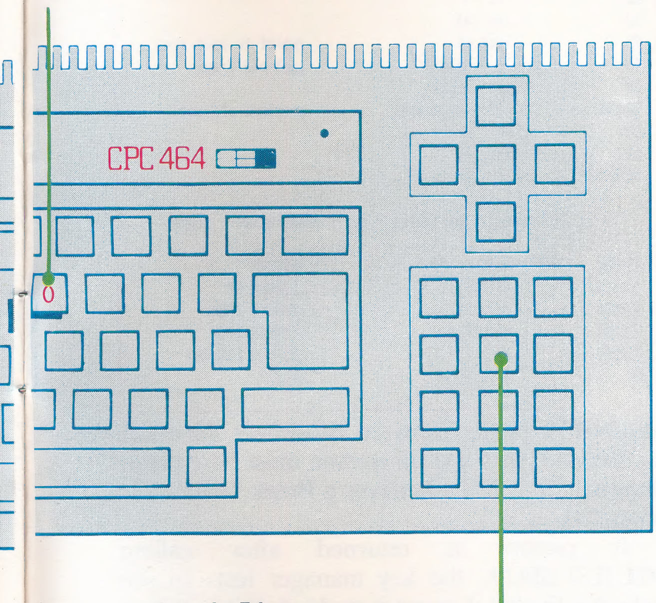
There are two key manager entries at the second level, KM_READ_KEY and KM_WAIT_KEY. KM_WAIT_KEY should be used to return the next keypress. If there is an entry in the keyboard buffer, then this is translated to either a character or an expansion token and returned immediately; otherwise, the routine waits for the next keypress and then returns its translated value. KM_READ_KEY also checks for the character from the keyboard buffer and if there is one then it is translated; otherwise, the routine immediately returns, flagging that the buffer is empty. KM_WAIT_KEY is therefore suitable for use when a program cannot continue until it has received input from the user — for example, when a choice from a menu is required. KM_READ_KEY should be used when the keyboard is being polled for a keypress.

The third level has two corresponding entries — KM_READ_CHAR and KM_WAIT_CHAR. The only difference between these and the second level entries is that only characters may be returned. If an expansion token is encountered, then the corresponding expansion string is read out character by character before the next entry from the keyboard buffer is examined.

Useful Addresses

&BB27	KM-SET-TRANSLATE	set a normal entry in the key translation table
&BB2A	KM-GET-TRANSLATE	read current normal entry in the key translation table
&BB2D	KM-SET-SHIFT	set entry in translation table for shifted key
&BB30	KM-GET-SHIFT	read entry in translation table for shifted key
&BB33	KM-SET-CONTROL	set entry in translation table for control key pressed
&BB36	KM-GET-CONTROL	read entry in translation table for control key pressed

Very Impressive
The firmware maintains a key-state map that registers the keys pressed at the time of the last general-purpose ticker interrupt (every 50th of a second). Key values are stored in a buffer and the firmware lets you either test for a keypress or wait for one



Expansion Tokens
ASCII codes &80 to &9F are treated as 'expansion tokens' by the firmware. These codes are assigned by default to the keys on the numeric keypad. Expansion strings can be assigned through firmware to any of these tokens. Such strings can be up to 255 characters in length and may contain control codes

THE PUTBACK CHARACTER

The key manager has a one-byte buffer in addition to the keyboard buffer, known as the putback character. This is checked before the main buffer and if it contains a character, it is given priority. This allows a character to be effectively inserted at the head of the buffer which could be used, for example, to flag that an event had occurred. The putback character is overwritten by the key manager when a break is detected, because the break token is inserted as the putback character.

The diagram shows the various levels at which a character may be returned from the key manager.

Each key on the keyboard has three repeat parameters associated with it. The first determines whether or not the key is allowed to repeat; the second specifies the delay between the detection of the keypress and when it is permitted to repeat; and the third is the delay between each individual repeat. Delays are specified in multiples of keyboard scans and therefore correspond to an interval of one-fiftieth of a second.

The current settings for delays may be determined using KM_GET_DELAY, or altered using KM_SET_DELAY. The delays apply to the entire keyboard so it is not possible to specify separate delays for different keys.

The parameter that determines whether or not a key is allowed to repeat is read and altered using the KM_GET_REPEAT and the KM_SET_REPEAT entries.

BREAKS

The Escape key is treated by the key manager in a different way to the other keys. Whenever it is pressed, the KM_TEST_BREAK indirection is called. This routine checks to see if Control and Shift are also pressed, in which case a reset is performed; otherwise the routine returns. By patching this indirection with a RET instruction it is therefore possible to prevent a reset occurring within a program.

There is also an event provided to deal with breaks. This event may be initialised, enabled or disabled using the firmware. By default, the event is disabled, so a special routine must be created that can be called whenever a Break keypress is detected.

If control is returned after calling KM_TEST_BREAK, the key manager tests to see whether the break event is enabled. If it is, then a character &EF is inserted into the buffer, and the break event is triggered. The marker is provided to allow the event to flush the keyboard buffer up to the point where the break was encountered.

This look at the key manager concludes our series on the Amstrad CPC operating system.

Key Translation

This listing provides two routines that allow a program to configure the keyboard to return a predefined set of codes during a program, and return them to their initial state on exit.

The first routine, Setkeys, should be called on entry to the program. This scans a table that contains a list of key numbers and what their translations should be set to. The current state of the keys is read and stored in the table before the new value is assigned to the key.

The second routine reads the original translations back out of the table and reassigns them to the keys.

The table here is set up to configure the cursor and escape keys for word processing use, but this may be tailored for individual applications

```
        ;SETKEYS
        get_tr: equ  #BB2A
        set_tr: equ  #BB27
        tend:   equ  #FE
21B329  setkey: ld   hl,tnormal          ;set norm trans table
7E      setlp:  ld   a,(hl)              ;get key number
FEFE            cp   tend                ;end of table?
C8              ret  z
23              inc  hl
46              ld   b,(hl)              ;get new trans
23              inc  hl
EB              ex   de,hl               ;save hl
F5              push af                  ;save key number
CD2ABB          call get_trans           ;read current trans
EB              ex   de,hl
77              ld   (hl),a              ;save old trans
23              inc  hl
EB              ex   de,hl
F1              pop  af
CD27BB          call set_trans           ;set new key
EB              ex   de,hl
18E9            jr   setlp               ;carry on
        ;
        ;GETKEYS
        ;
21B329  getkey: ld   hl,tnormal          ;restore norm table
7E      getlp:  ld   a,(hl)              ;get key number
FEFE            cp   tend
C8              ret  z
23              inc  hl
23              inc  hl                  ;point to save area
46              ld   b,(hl)              ;get old trans
EB              ex   de,hl
CD27BB          call set_trans           ;restore old one
EB              ex   de,hl
23              inc  hl
18F1            jr   getlp
        ;
        ;key translation tables - key - translation - store
        ;
421B00  tnorma: defb 66,27,0             ;ESCAPE
000B00          defb 0,11,0              ;CURSOR UP
020A00          defb 2,10,0              ;CURSOR DOWN
080800          defb 8,8,0               ;CURSOR BACK
010900          defb 1,9,0               ;CURSOR FORWARD
FE              defb tend                ;end
```

Special Codes

&80–&9F	Expansion tokens & 80F corresponds to expansion string 0, and &9F to expansion string 32
&E0–&FC	BASIC cursor control codes
&FD	Caps lock on/off toggle Reverses the current state of the caps lock
&FF	Shift lock on/off toggle Reverses the current state of the shift lock
&FF	Ignore token This character should be ignored if returned from the key manager

THE HOME COMPUTER ADVANCED COURSE

INDEX TO ISSUES 73 TO 84

A

AC converter 1585
A/D converters 1453, 1464, 1484, 1516, 1524
ALGOL 1587
Amsoft speech synthesiser **1469-1471**
Amstrad CPC 464/664
Background Beeper 1599
Go game 1455, 1472, 1497, 1533
interactive character game 1507-1508
MIDI interface 1632-1634, 1643-1645
operating system 1538-1540, 1558-1560, 1578-1579, 1598-1599, 1616-1619, 1638-1640, 1657-1659, 1678-1680
Spreadsheet program 1595, 1609, 1627, 1652
Amstrad CPC 664 **1490-1491**
Amstrad CPC 6128 **1629-1631**
Apple Laserwriter **1529-1531**
Apple Macintosh 1581-1583
Arrays 1462, 1555
Artificial intelligence **1444-1445, 1461-1463, 1481-1483**
Atari 520ST **1549-1551**
Attenuator circuits 1584

B

BASIC 1445
BBC Micro
Go game 1454, 1472, 1496, 1514
interactive character game 1507-1508
Spreadsheet program 1594, 1609, 1627, 1652
BBC+ **1449-1451**
Bitstik controller 1522
Boulder dash **1480**

C

COBOL 1586, **1663-1665**
Colossal Cave 1441
Commodore 64
Go game 1456, 1473, 1496, 1532
interactive character game 1507-1508
Spreadsheet program 1566, 1609, 1627, 1653, 1667
Computer aided design **1521-1523**
Computers in gambling **1601-1603, 1621-1623, 1649-1651,1672-1673**
Computers in industry **1510-1511, 1521-1523, 1541-1543**
Computers in publishing **1561-1563, 1581-1583**
Computer integrated manufacturing 1510
Computer numerically controlled machines 1511, **1541-1543**
Control structures 1486
Coursewinner 1601-1602
CP/M 1491

D

D/A converters 1453, 1464, 1484
Decision trees 1462
Digital meters 1452
Dog & Bucket murder mystery game 1674-1677
Dynabook 1671

E

Early computer languages **1586-1588**
EDSAC 1586
Eliza program 1441
Epson SQ-2000 printer **1610-1611**
Eratosthenes' Sieve 1504
Eurisko program 1481
Events 1579, 1598

F

Fabry, Charles 1502
Football Manager **1500**
FORTH 1445, **1446-1448, 1475-1477, 1486-1488, 1504-1505, 1535-1537, 1555-1557, 1567-1568**
FORTRAN 1586, **1613-1615, 1635-1637, 1654-1655**
Frankie Goes To Hollywood **1515**

G

GEM operating environment 1550-1551
Go game **1454-1457, 1472-1474, 1495-1497, 1512-1514, 1532-1534, 1546-1547**

H

The Hobbit 1441-1442
Hopper, Grace 1586, 1613
HRX system 1522
Hulk I and II 1603

I

IBM PC/AT **1569-1571**
Inheritance hierarchies 1462
Ink jet printers 1610-1611
Interactive character games **1441-1443, 1466-1467, 1492-1494, 1506-1508, 1526-1527, 1544-1545, 1575-1577, 1596-1597, 1604-1605, 1624-1625, 1646-1648**
Interface 1 1458-1459, 1478-1479, 1498-1499
Interrupts 1578
Island Logic Music System **1620**

K

Knowledge representation 1461-1463
Kuma FORTH 1535

L

Lascar LMM 100 meter 1452
Laser disc systems 1589-1591
LISP 1444-1445
LOOPS 1445

M

Mach Zehnder interferometer 1502
Macintosh ImageWriter 1531
Man-machine hybrids 1483
Maplin Precision Gold M-5010 meter 1452
Matchbox package 1523
Mauchly, John 1586
Microprocessors **1661-1662**
MG-1 workstation 1523

N

Namal Type & Talk **1469-1471**

THE HOME COMPUTER ADVANCED COURSE

INDEX TO ISSUES 73 TO 84

O

Ohms converter 1585
Operating systems **1458-1459, 1478-1479, 1498-1499, 1518-1520, 1538-1540, 1558-1560, 1578-1579, 1598-1599, 1616-1619, 1638-1640, 1657-1659, 1678-1680**
Optical computers **1501-1503**

P

PageMaker package 1581-1583
PARC 1670-1671
PASCAL 1445
Payoff formula 1622
Perot, Alfred 1502
Pioneer PX-7 **1589-1591**
POPLOG 1445
Print-Technik video digitiser **1641-1642**
Probability theory 1649
PROLOG 1444-1445
Prospector program 1481

R

Read/write head 1460
Real-time 1460
Record 1460
Recoverable error 1460
Recursion 1468
Redundancy 1468
Refresh 1468
Register 1468
Relation 1468
Relational database 1489
Resolution 1489
Reverse Polish notation 1475-1476, 1489
RGB 1489
Ring network 1509
Robotics 1509
Rockford's Riot **1480**
ROM 1509
Roundoff error 1509
RS232C 1528
Run-time 1528

S

S-100 bus 1548
Sampling 1528
Sawtooth 1528
Scrolling 1548
SDLC 1548
Search algorithm 1580
Sector 1580
Semeai 1454
Semantic nets 1462
Semiconductor 1580
Sequential access 1580
Serial input/output 1600
Server 1600
Servo motor 1600
Set 1600
Shell sort 1612
Shift register 1612
Simulation 1612
Sine wave 1612
Sherlock 1507
Shicho 1474
Short Code 1587
Sinclair Spectrum
Go game 1457, 1473, 1497, 1534
interactive character game 1507-1508
network system 1458-1459, 1478-1479, 1498-1499, 1518-1520
Spreadsheet program 1593, 1609, 1627, 1653, 1668
Slave 1628
SMALLTALK 1670-1671
Soar ME-531 meter 1452
Soft keyboard 1628
Software 1628
Sorting 1628
Source code 1660
Spectrum Shadow ROM Disassembly 1478
Speech synthesisers 1469-1471
Split screen 1660
Spool 1660
Spreadsheet programming **1564-1566, 1592-1595, 1608-1609, 1626-1627, 1652-1653 1666-1668**
Sprint Formula 1602
SP0256 chip 1470
Square wave 1660
Stack 1475, 1660
Standardisation 1669
Star network 1669
Stepper motor 1669
Storage device 1669
Stream 1669
Suspect 1441-1443, 1466

T

Thickness 1533-1534

U

UART 1600
Ultra-intelligent machines 1482

V

Valhalla 1442-1443
Video digitisers 1641-1642
Votrax SC01A chip 1469

W

WIMP systems **1670-1671**
Workshop digital multimeter **1452-1453, 1464-1465, 1484-1485, 1516-1517, 1524-1525, 1552-1554, 1572-1574, 1584-1585, 1606-1607**
Workshop MIDI interface **1632-1634, 1643-1645**

Z

Z5 Horse Race Forecast 1602

7135 chip 1464-1465, 1484-1485, 1516-1517, 1524-1525
74LS00 chip 1607